

# Combinational Logic Continued; Memory and Sequential Circuits

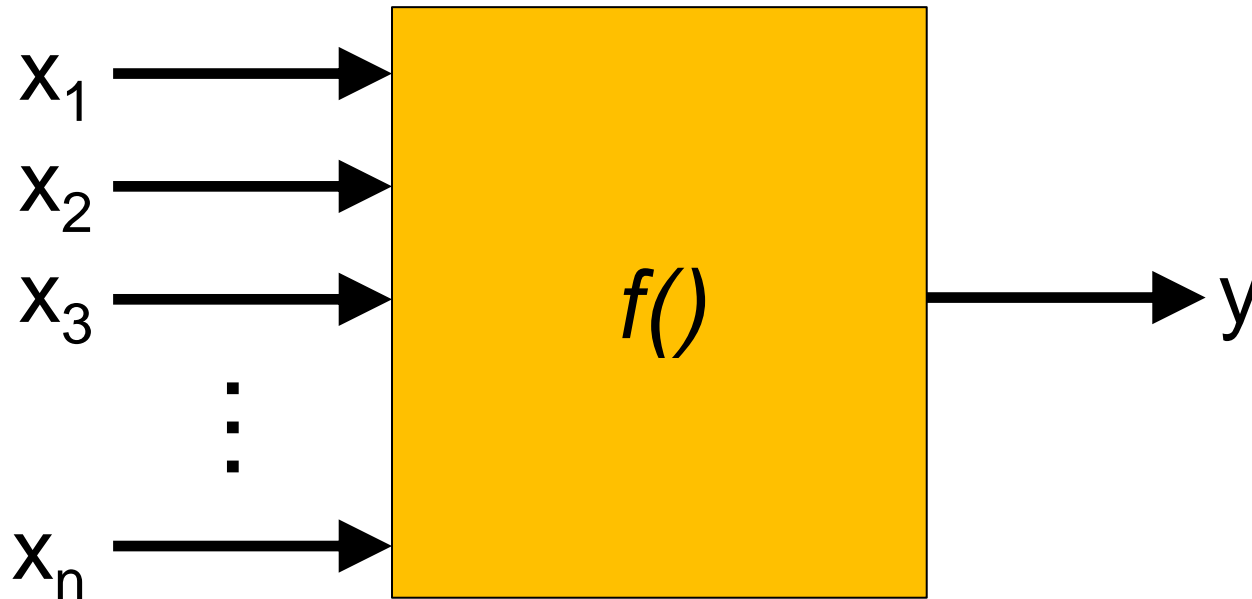
For use in CSE6010 only

Not for distribution

# Boolean Functions

Goal: Design a circuit to implement an *arbitrary* Boolean function on  $n$  binary inputs:

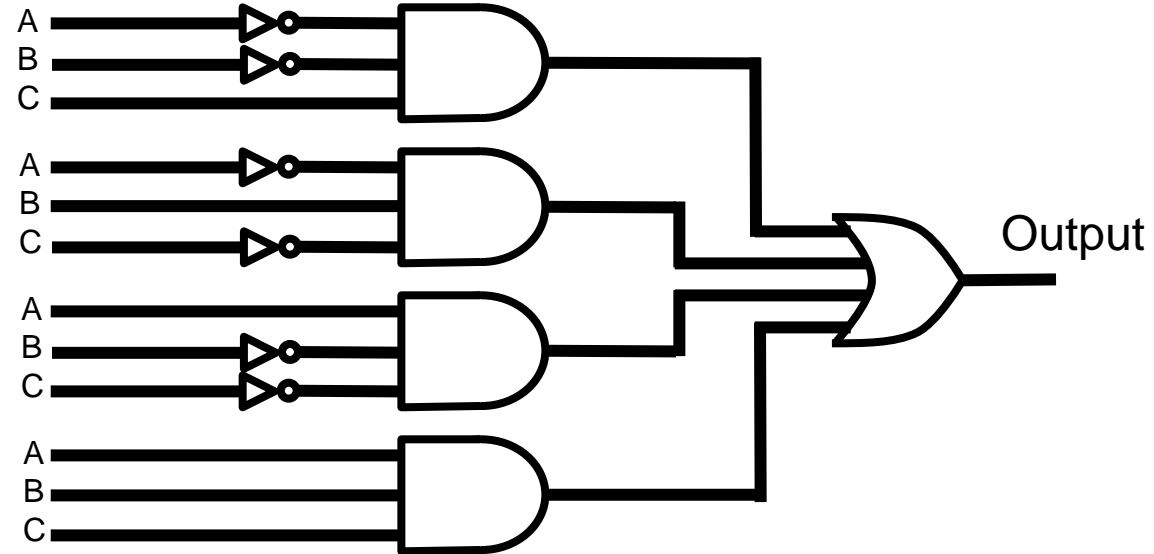
$$y = f(x_1, x_2, \dots, x_n)$$



# Full Adder: Sum Output

$A_i$	$B_i$	$C_i$	$S_i$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

$$S_i = \bar{A}_i \bar{B}_i C_i + \bar{A}_i B_i \bar{C}_i + A_i \bar{B}_i \bar{C}_i + A_i B_i C_i$$



- Any logic function can be completely specified by a truth table
- Any truth table can be converted to a Boolean equation
- Any Boolean equation can be implemented by a circuit
- Therefore, any logic function can be realized with a circuit

Caveats: (1) In practice, number of inputs (fanin), outputs (fanout) for a gate limited, but there are ways around this; (2) above approach may not yield optimal circuits in terms of number of gates or delay

# Canonical Sum of Products Form

$A_i$	$B_i$	$C_i$	$S_i$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

minterm

- Every row whose output is one gets a term; these terms are added (OR'd)
- Each term is the product (AND) of each input or its complement (whichever is 1) with the other inputs (or their complements)

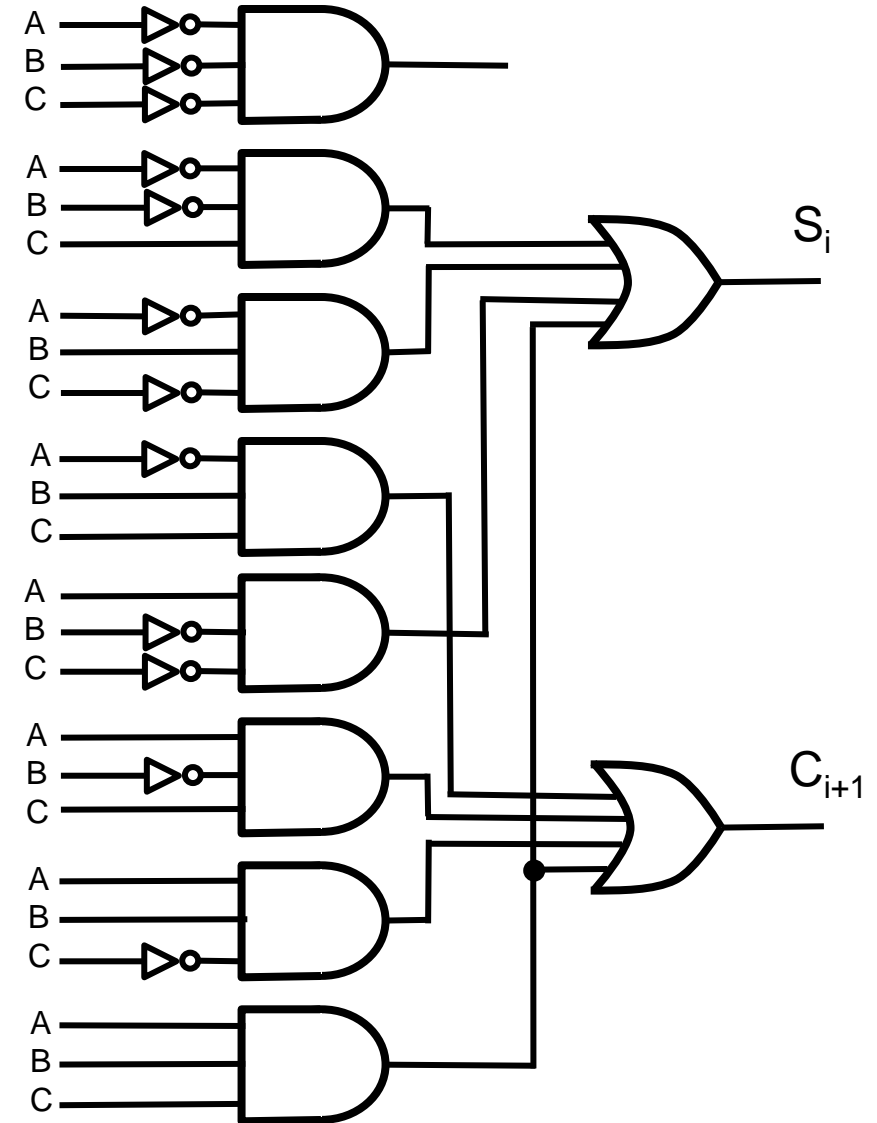
$$S_i = \bar{A}_i \bar{B}_i C_i + \bar{A}_i B_i \bar{C}_i + A_i \bar{B}_i \bar{C}_i + A_i B_i C_i$$

- Canonical sum of products representation
  - Boolean equation is a sum (i.e., *or*) of product terms (i.e., *and*)
  - Each product term includes all input variables, either complemented or not complemented (aka minterm)
- One can often derive simpler Boolean equations which typically leads to fewer gates (not discussed here)
  - Methods exist to minimize the number of gates

# Full Adder Circuit

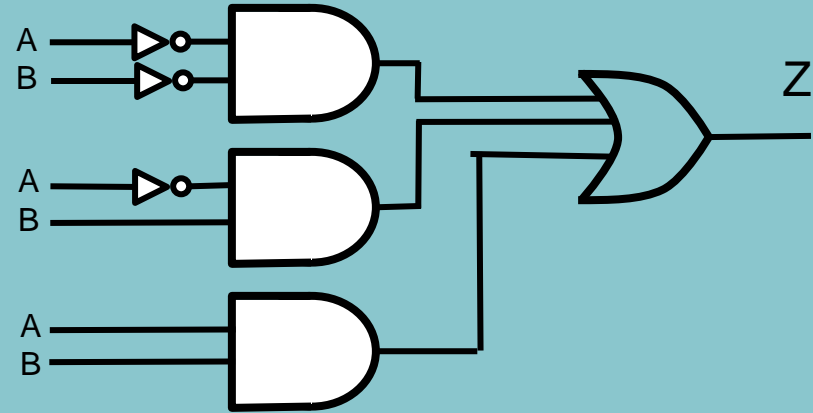
$A_i$	$B_i$	$C_i$	$S_i$	$C_{i+1}$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

- $2^3=8$  different input combinations
- 4 lead to  $S_i=1$
- 4 lead to  $C_{i+1}=1$
- 1 overlaps; 1 results in  $S_i=C_{i+1}=0$



# Examples: Circuits $\Leftrightarrow$ Truth Tables

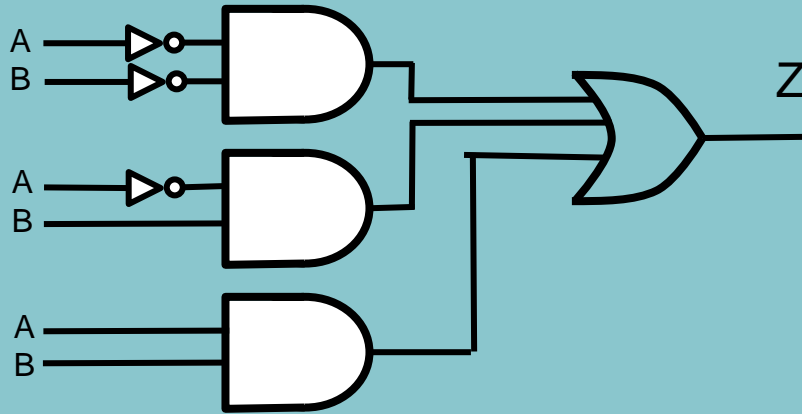
Write the truth table that corresponds to the circuit.



Draw the circuit that corresponds to the truth table. (Use the method we just considered even though you may realize a simpler circuit.)

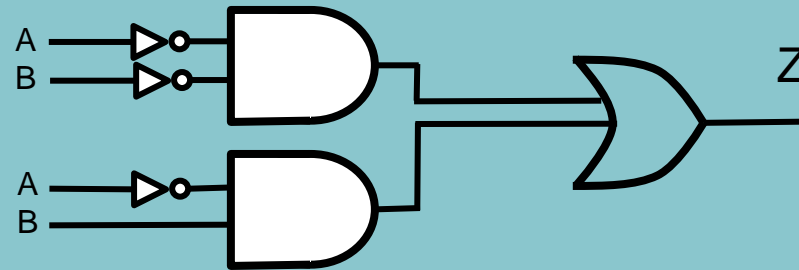
$A_i$	$B_i$	$Z$
0	0	1
0	1	1
1	0	0
1	1	0

# Examples: Circuits $\Leftrightarrow$ Truth Tables



$A_i$	$B_i$	Z
0	0	1
0	1	1
1	0	0
1	1	1

$A_i$	$B_i$	Z
0	0	1
0	1	1
1	0	0
1	1	0



You may have realized it's simpler just to invert  $A_i$ .

# DeMorgan's Law

A or B is equivalent to not (not A and not B)

E.g.: “The pen is red or blue” is equivalent to

“it is not the case that the pen is both not red and not blue”

$$A+B = \overline{\overline{A}\overline{B}}$$

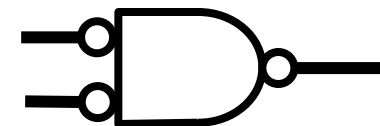
Start with OR function:

1. Complement inputs
2. Perform an AND
3. Complement output

A	B	A+B	$\overline{A}$	$\overline{B}$	$\overline{A}\overline{B}$	$\overline{\overline{A}\overline{B}}$
0	0	0	1	1	1	0
0	1	1	1	0	0	1
1	0	1	0	1	0	1
1	1	1	0	0	0	1

This: 

is equivalent to



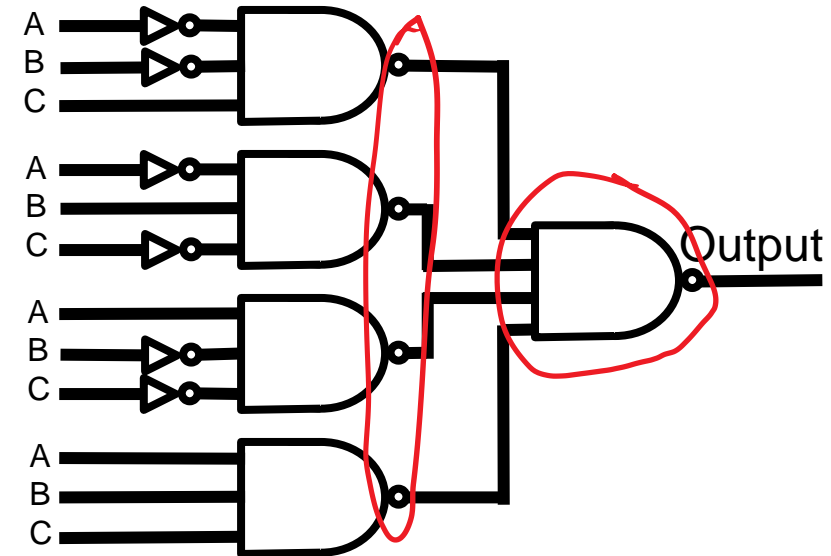
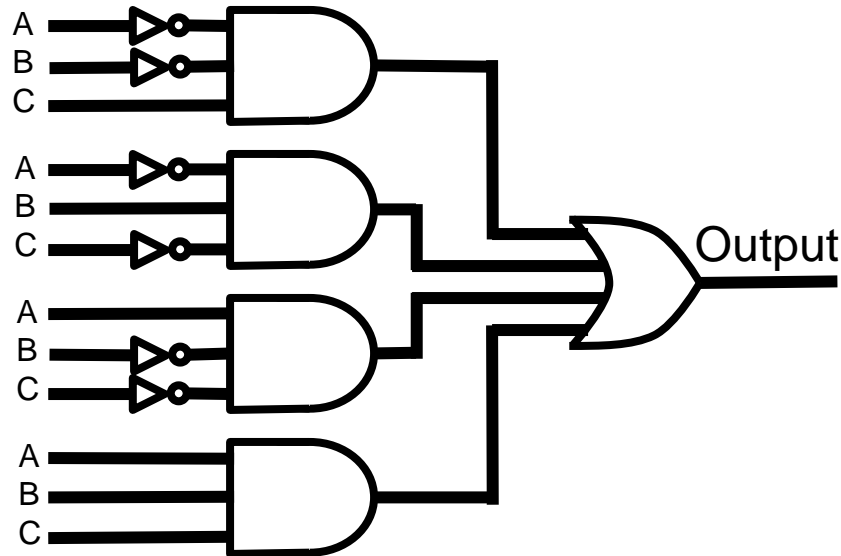
- One can create an equivalent circuit by (1) converting an OR gate into an AND gate and (2) complement the gate's inputs and outputs
- It can be shown: one can also create an equivalent circuit by (1) converting an AND gate to an OR gate and (2) complementing the gate's inputs and outputs



# Applying DeMorgan's Law

$$\text{Output} = \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C} + ABC$$

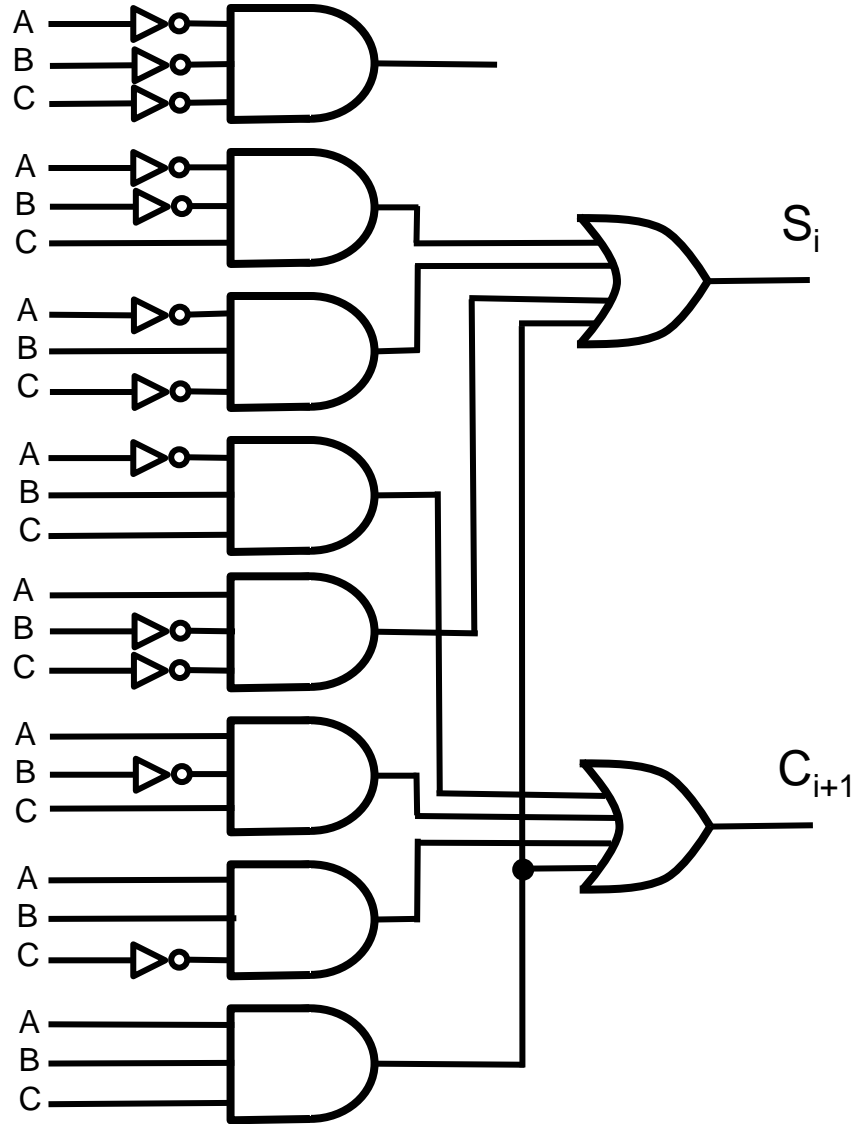
Replace OR gate with a NAND gate with complemented inputs



- We can implement any logic circuit using only NAND gates
  - Inverter is a NAND gate with a single input (equivalent to  $X \text{ nand } X$ )
  - Recall our CMOS circuits readily implemented NAND (and NOR), but AND (and OR) required an extra inverter
  - NAND implementation faster, requires less silicon area than AND/OR

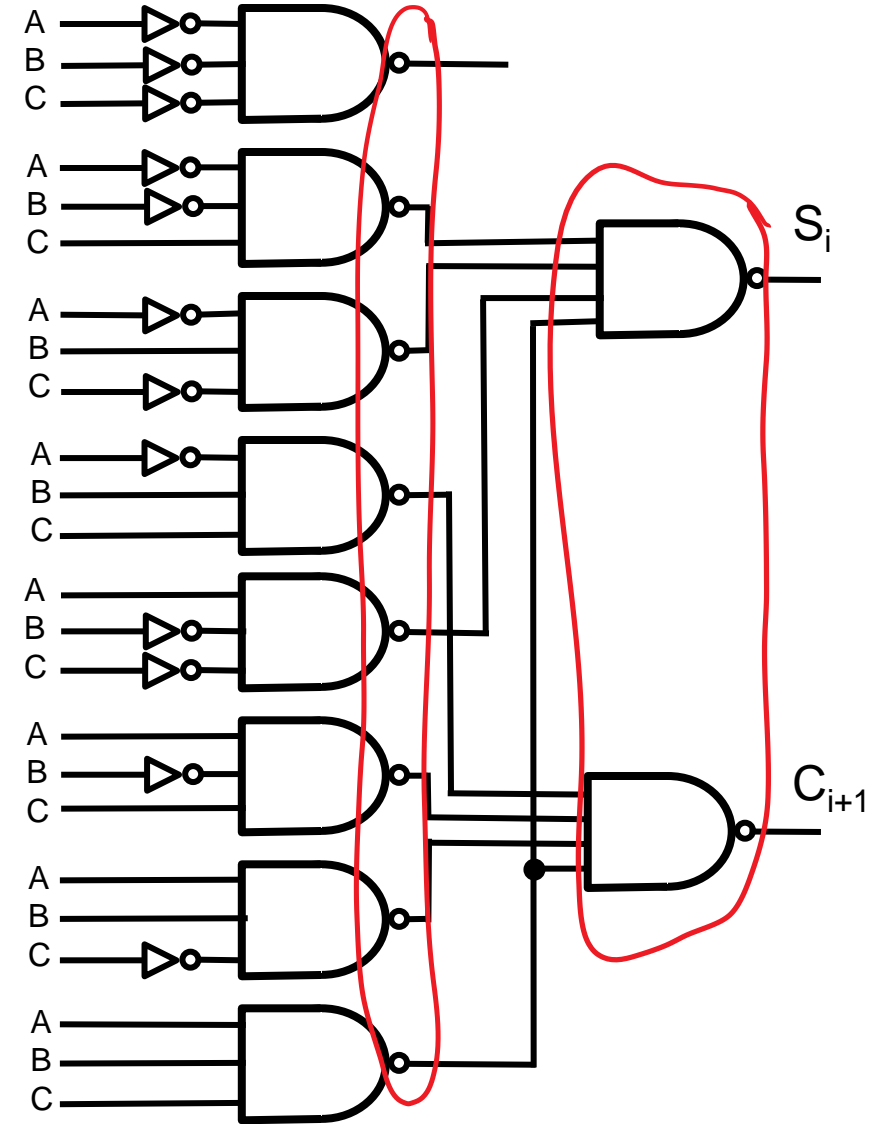
# NAND-Only Full Adder Circuit

Original

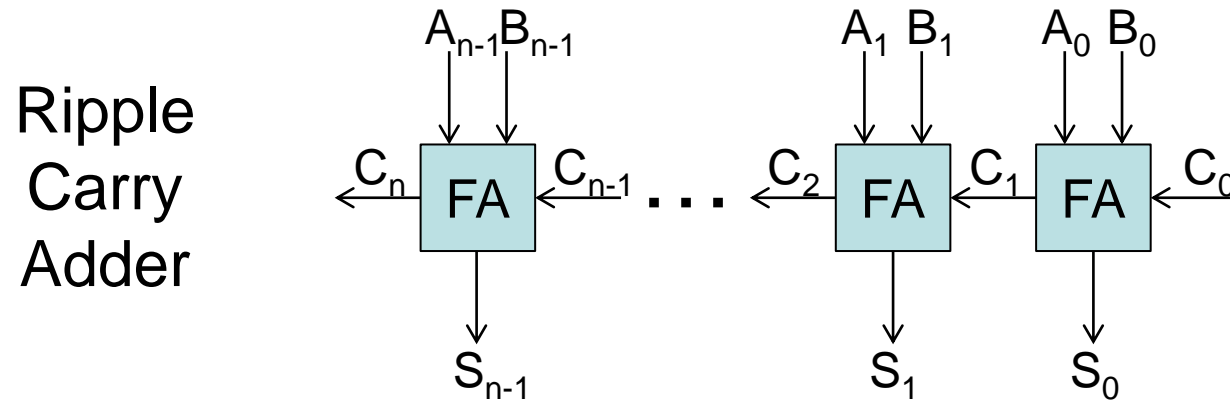


$A_i$	$B_i$	$C_i$	$S_i$	$C_{i+1}$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

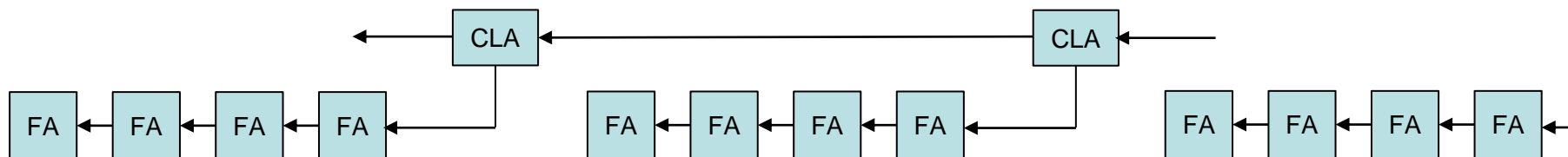
NAND-only



# Fast Addition



- Assume delay of  $\tau$  for a signal to travel through a single gate
- How many gate delays for one addition (Big-O)?
- Carry Lookahead (CLA) adder
  - Cluster full adders into groups (e.g., 4 full adders per group)
  - CLA circuit computes carry out of a group of full adders quickly
  - CLA tree can perform N-bit addition in  $O(\log N)$  gate delays

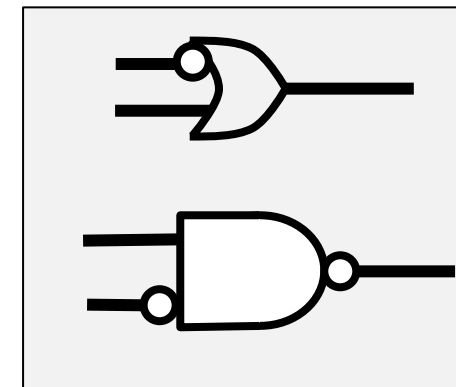
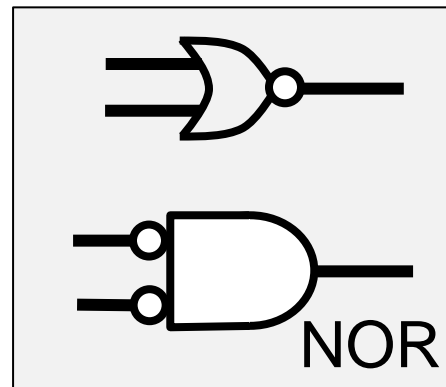
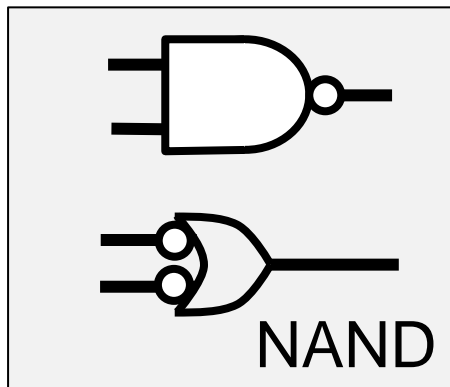


# Useful Boolean Algebra Laws

A, B, and C are Boolean variables

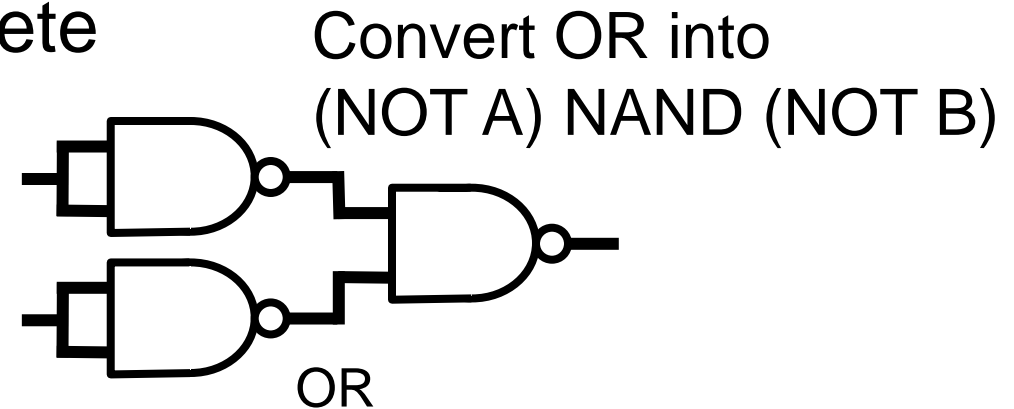
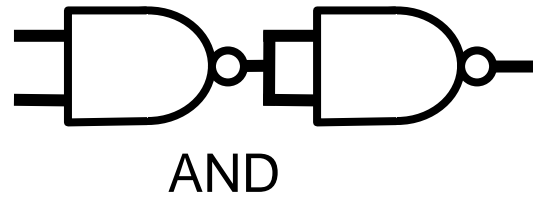
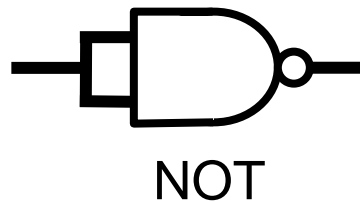
Law	AND	OR
Identity	$A \cdot 1 = A$	$A + 0 = A$
Null	$A \cdot 0 = 0$	$A + 1 = 1$
Idempotent	$AA = A$	$A + A = A$
Complement	$\overline{\overline{A}} = A$	
Commutative	$AB = BA$	$A + B = B + A$
Associative	$(AB)C = A(BC)$	$(A+B)+C = A+(B+C)$
De Morgan	$\overline{AB} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A} \overline{B}$

De Morgan's Law: the gates within each box are equivalent



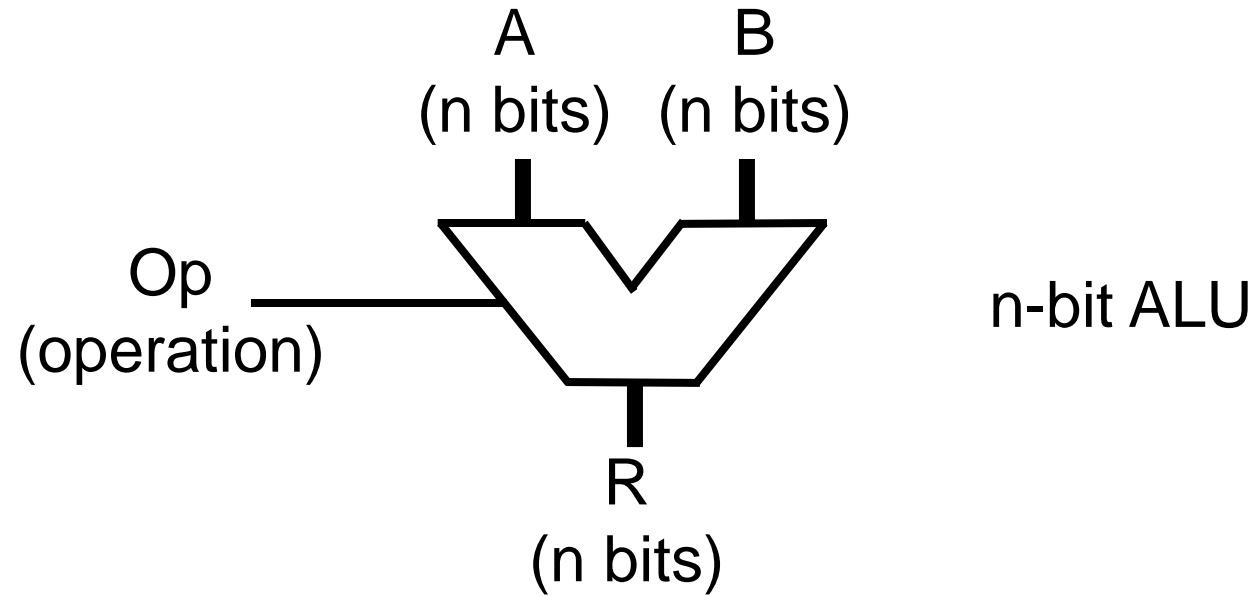
# Logical Completeness

- For any Boolean logic function a circuit realization can be obtained using AND, OR, NOT gates
- The set {AND, OR, NOT} is said to be **logically complete** (sufficient to implement any Boolean function)
- The set {NAND} is also logically complete



- One can show that {NOR} is logically complete
- Logical completeness + memory allow one to create a machine that can compute anything that is computable (Turing machine)

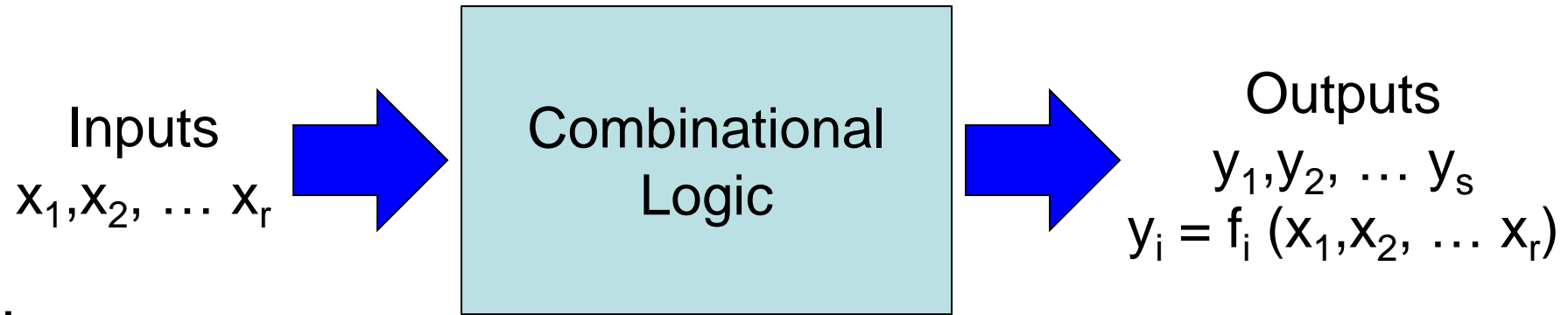
# Arithmetic / Logic Unit (ALU)



- Generalization of adder circuit that can perform
  - Arithmetic (addition, subtraction)
  - Logic operations (AND, OR, NOT)
- More sophisticated ALUs can perform multiplication, division; however, these operations take much longer to perform

# Combinational Logic Circuits

The circuits discussed so far are called combinational logic circuits



## Properties

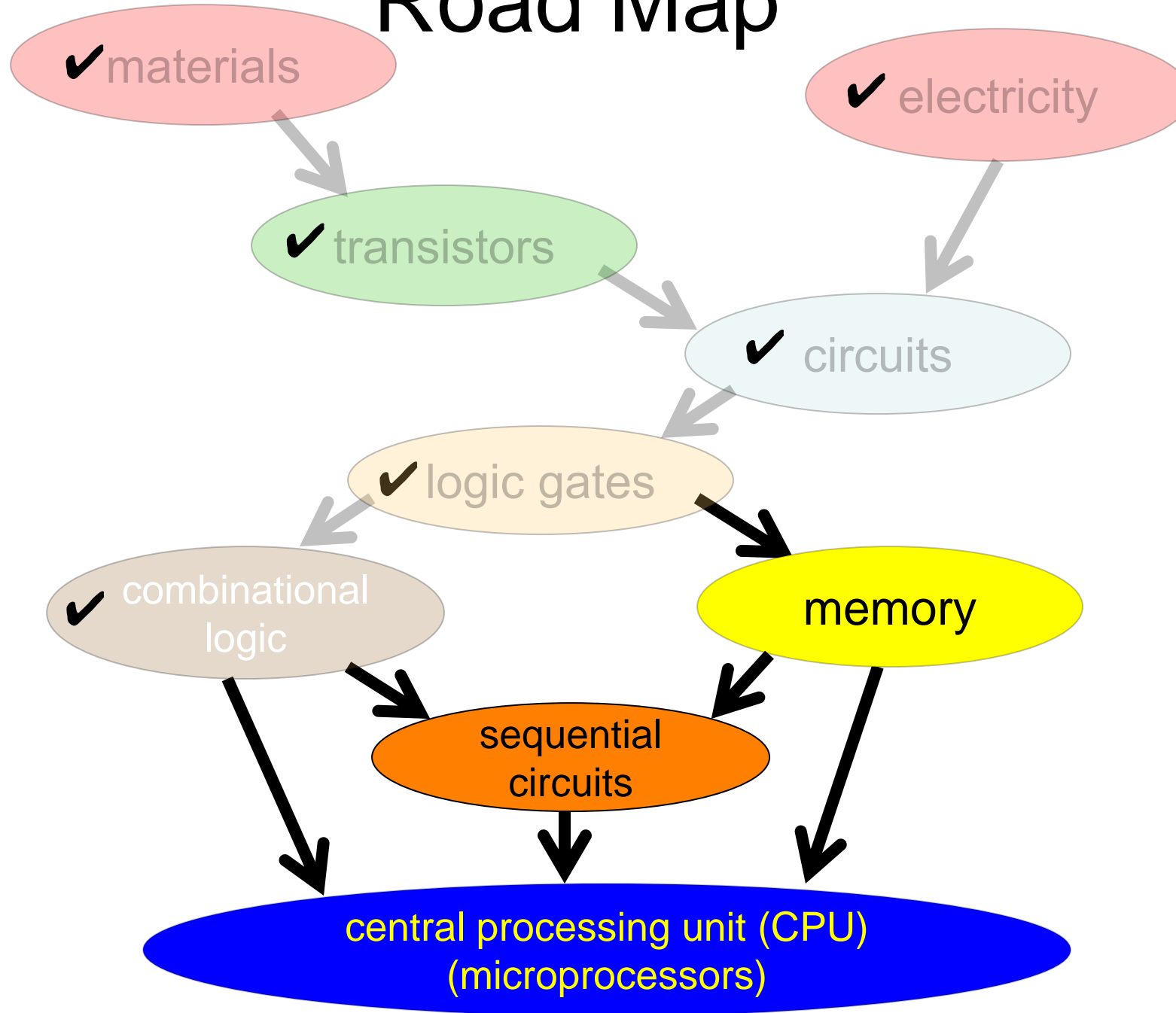
- No feedback signals
- Output only a function of current inputs; change the inputs and the outputs immediately (ignoring circuit delays) change
- **No memory**; output independent of previous operations and inputs

# Summary

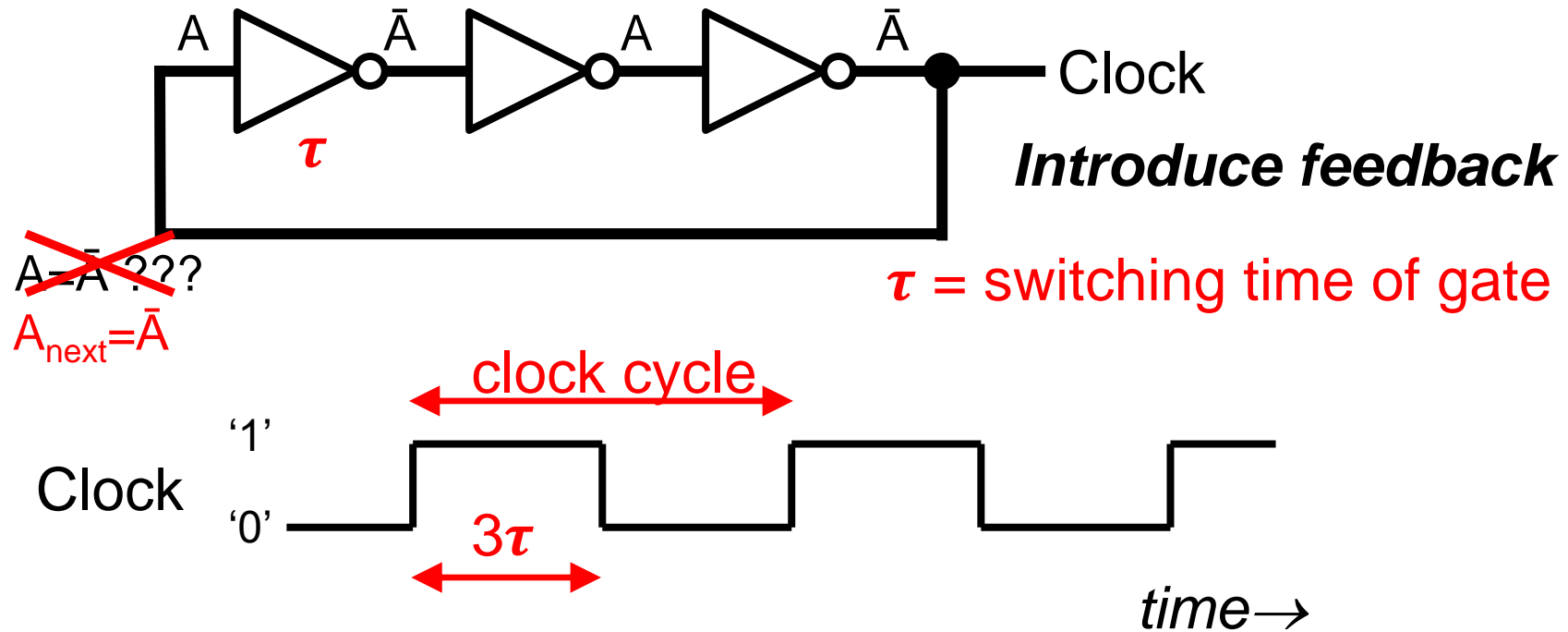
- Boolean Algebra enables us to create arbitrary Boolean functions from logic gates (logical completeness)
- Logic functions enable us to implement basic computational operations (e.g., arithmetic)



# Road Map

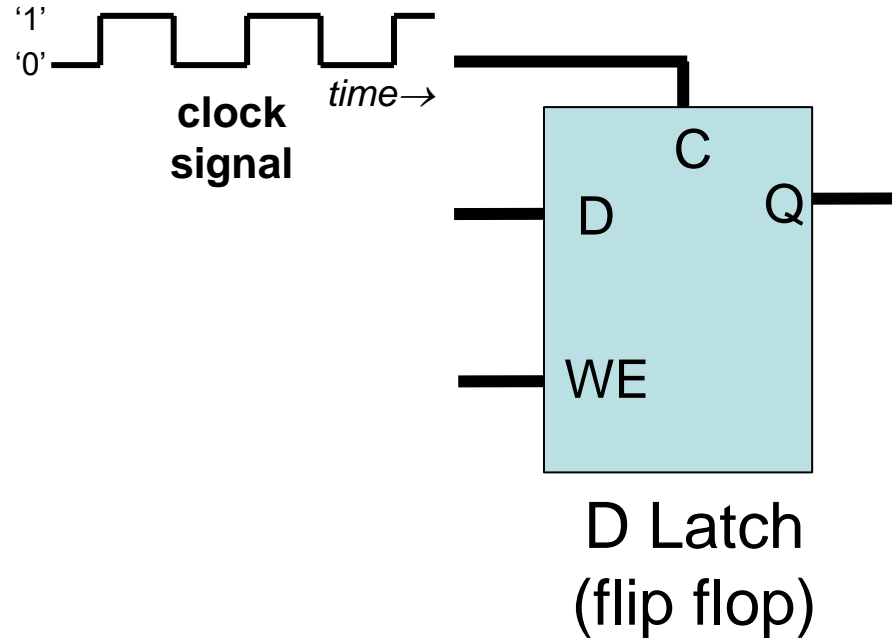


# Clock Signal



- Clock signal used in all CPUs today
  - “Advertised” clock rate indicates the frequency of this clock signal, e.g. 1.0 GHz, or 1 nanosecond ( $10^{-9}$  second) clock cycle time
- Used in synchronous circuits (state machines)
  - Asynchronous circuits do not use a clock

# Memory: D Latch (aka D Flip Flop)





Stores 1 bit of information

Inputs:

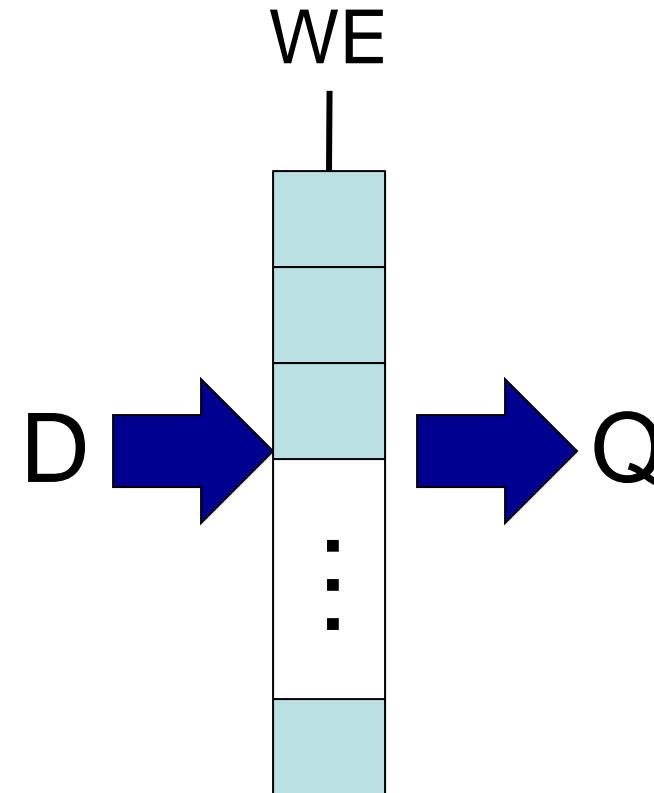
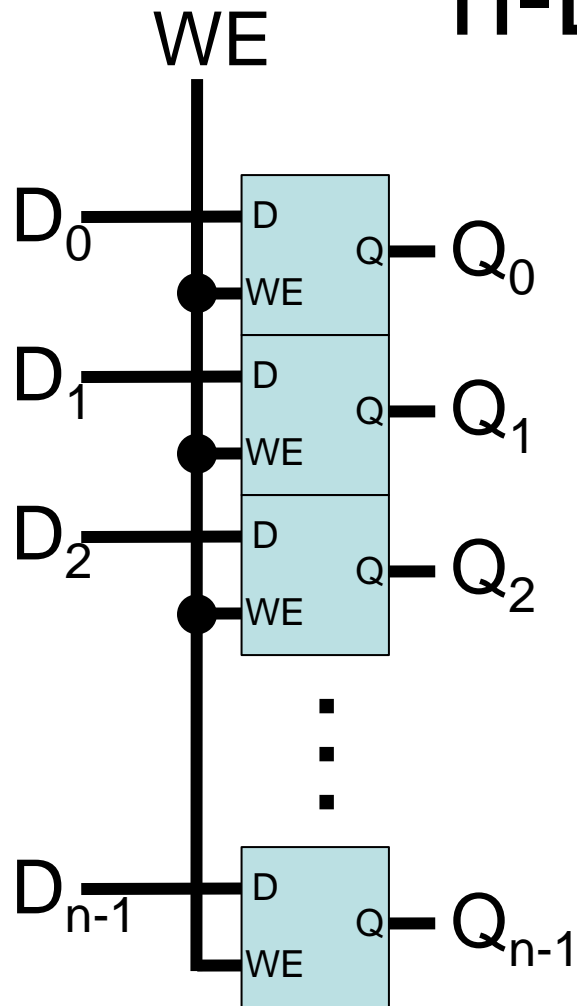
- D: data input
- WE: write enable
- State only changes on rising edge of clock (C)

Output:

- Q: data stored

C	WE	D	Q	operation
	1	d	d	write D into flip flop storage
	0	-	q	maintain current state (no action)
anything else	-	-	q	maintain current state

# n-Bit Register

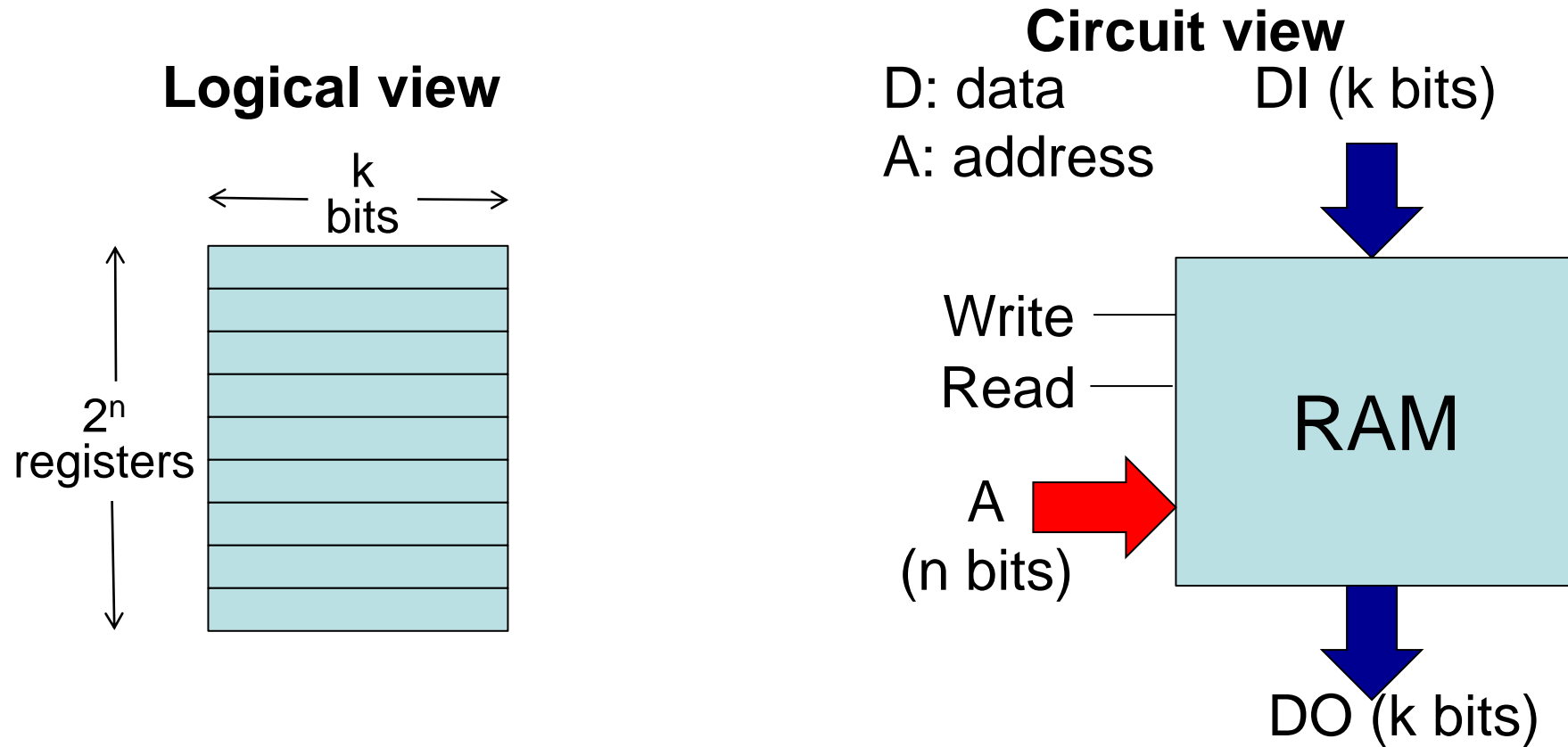


Clock signal not shown

A register is an array of D flip flops

- $WE=1$ : write data (D) into register (next rising edge of clock)
- $WE=0$ : register retains data last written into it

# Random Access Memory (RAM)

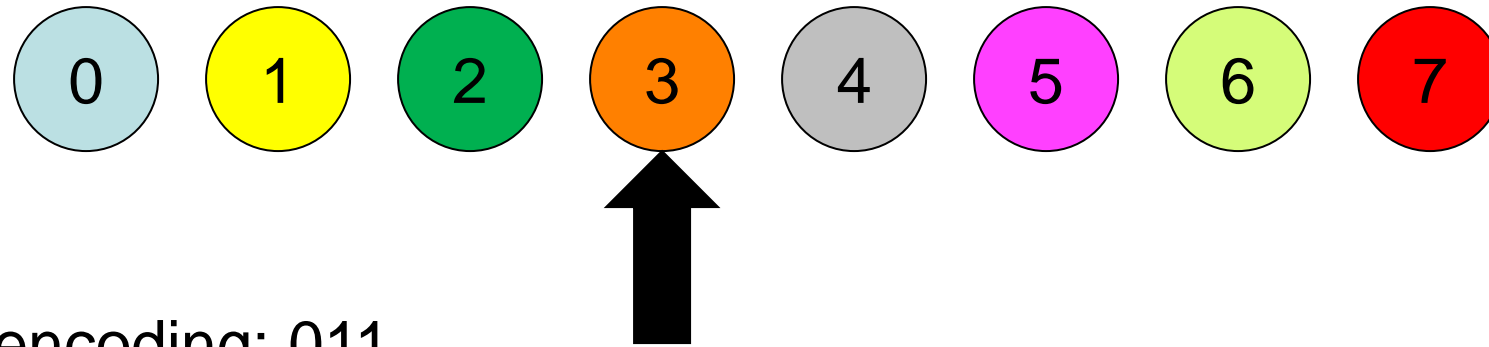


- Array of registers (2D array of D flip flops)
- $2^n$  registers each holds a single “chunk” of data (k bits)
- Can read or write a single “chunk” at a time
  - Control lines: Read, Write (assert to read/write data at address A)
- Address: n-bit binary number indicating which chunk ( $2^n$  chunks)

# Binary and Unary Encoding

- Given a set of  $N$  items:  $0, 1, 2, \dots, N-1$
- We want to select one of them

Example,  $N=8$



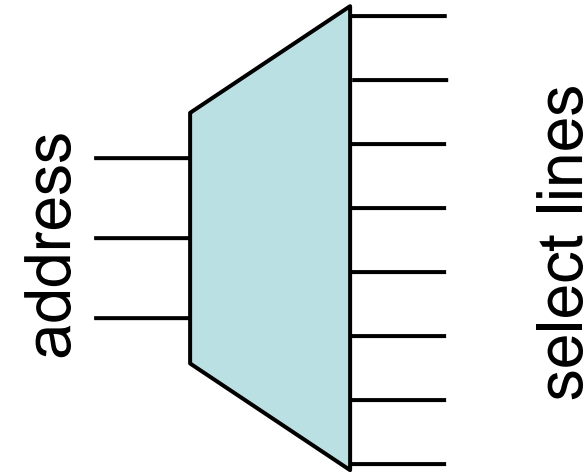
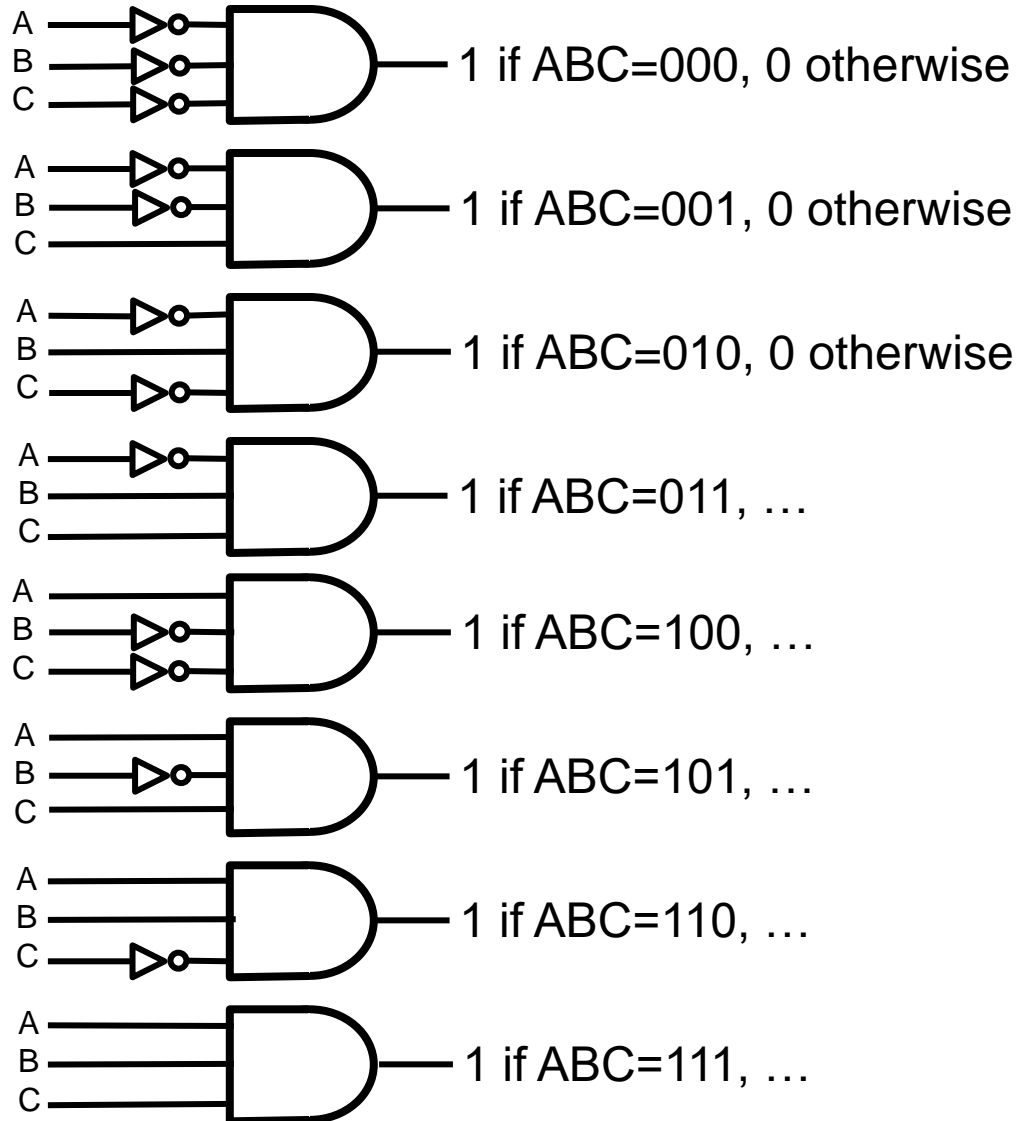
Binary encoding: 011

Unary encoding:

0    0    0    1    0    0    0    0

**Decoder circuit** converts binary encoding to unary

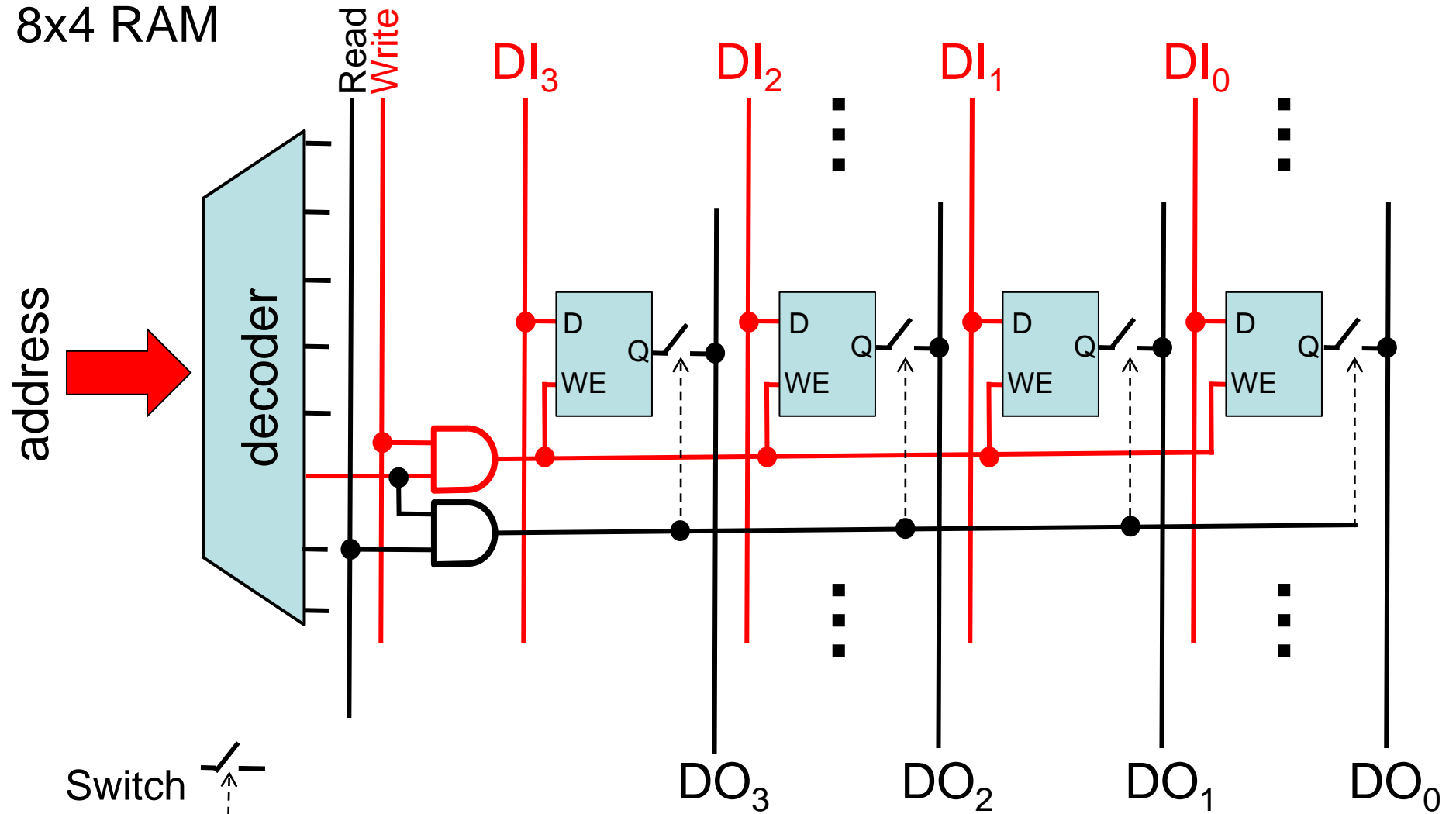
# Decoder: Circuit to Select a Register from a Binary Number




- $n$  inputs,  $2^n$  outputs
- Exactly one output is asserted (1) for any possible input (other outputs not asserted, i.e., are 0)
- Used in memory components to select a memory location

# RAM Implementation

8x4 RAM



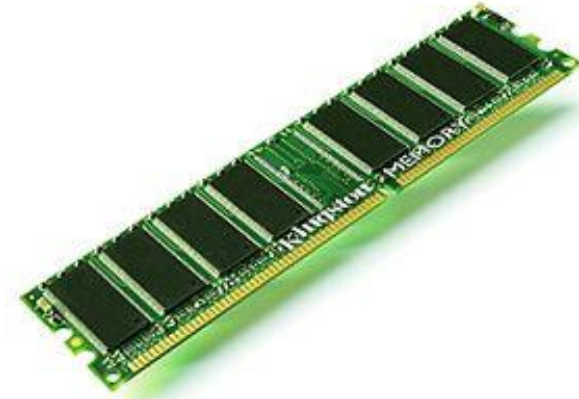
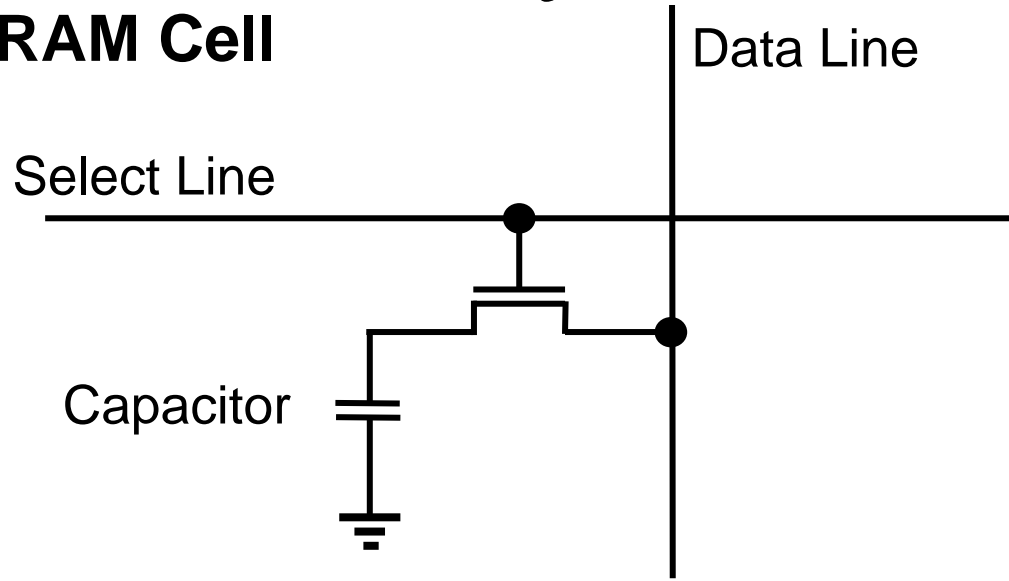
Switch   
C = 0 open  
C = 1 closed

**Static RAM:** implement each D flip flop with gates



# Dynamic RAM

## DRAM Cell



DRAM circuit board

- Interface similar to that defined already (addr, data, rd/wr)
- Capacitor has the ability to store charge
  - 0: no charge stored on capacitor
  - 1: charge stored on capacitor
- Reading the cell discharges the capacitor (if it held a charge)
- Charge “leaks” from capacitor
- Additional circuitry is required to periodically “refresh” data (read data and immediately write it back)

# SRAM vs. DRAM

## Static RAM

- Retains data as long as the circuit continues to receive power
- Fast access time for read and write operations
- Requires more Si area than DRAM
- Used in processor registers, register file, fast memory (cache)

## Dynamic RAM

- Charge must be “refreshed” to retain data, even with continuous power
- Slower access time for read and write operations
- Very compact, high density (more bits/unit Si area)
- Used in “main memory” for processor

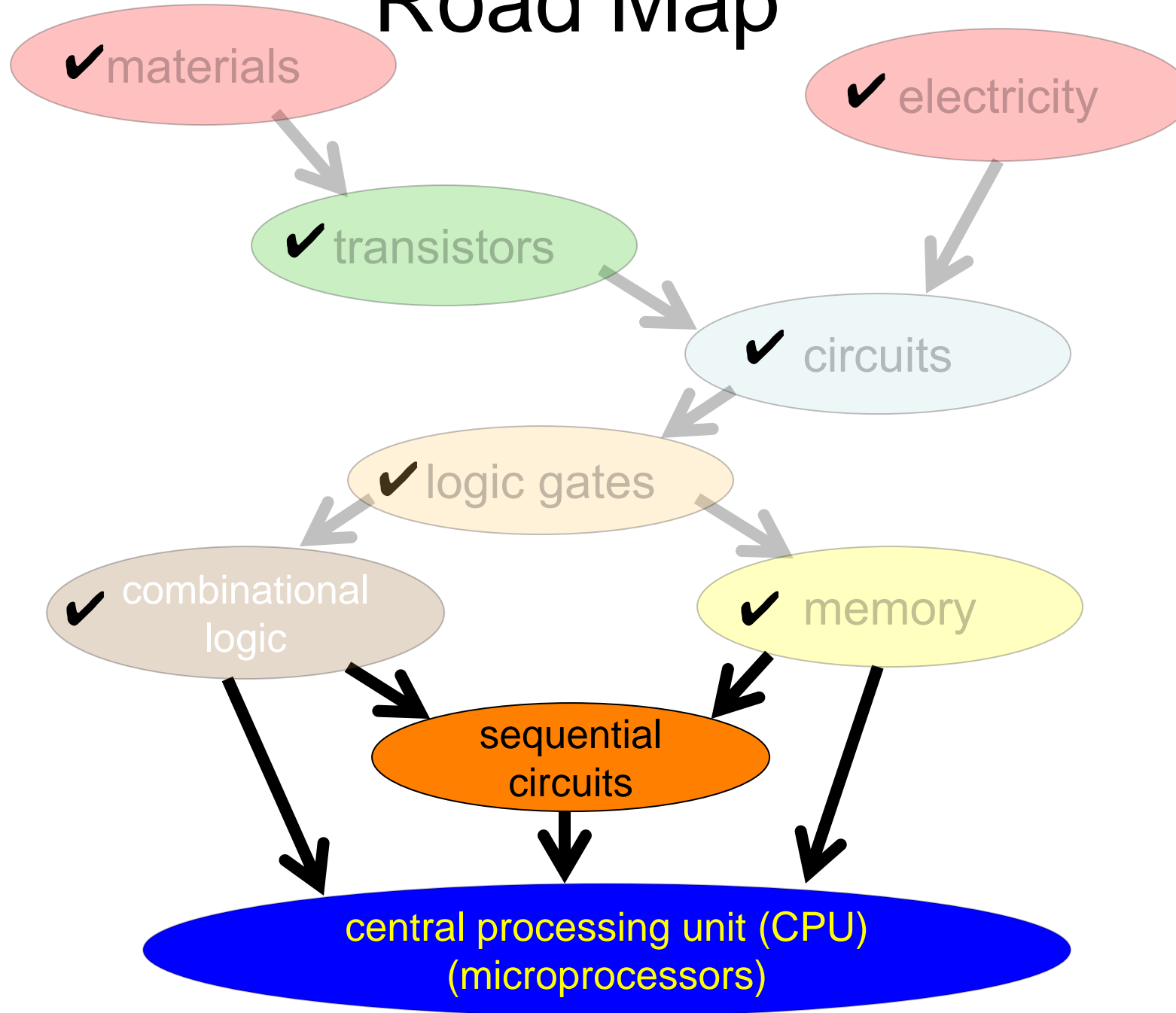
Other types of memory:

ROM, SSD, PROM, Flash, disk/tape: retain data w/o power

# Observations

- Memory implemented as  $N \times M$  array of bits
  - $N$  always a power of 2 (binary address)
  - $M$  typically a power of 2 also (e.g., 32 or 64 bits) sometimes called a “word”
- Computers designed to only operate on a few specific sizes of data; sizes which can be easily packed into  $M$ -bit words
  - 8 bits (byte): character
  - 16 bits: short integer
  - 32 bits: integer, single precision floating point
  - 64 bits: double precision floating point

# Road Map



# Sequential vs. Combinational Circuits

- Combinational Circuits
  - Outputs only depend on current input
  - Output same, independent of time
  - Circuit has no memory (no feedback)
- Sequential Circuits (aka state machines)
  - Contains memory within the circuit
  - Output can depend on **past** inputs (history)
  - Mealy machine: Output depends on current state and current inputs (not used here)
  - Moore machine: Output depends on current state only

# Combinational vs. Sequential

Two types of “combination” locks



## **Combinational**

Success depends only on the **values**, not the order in which they are set.



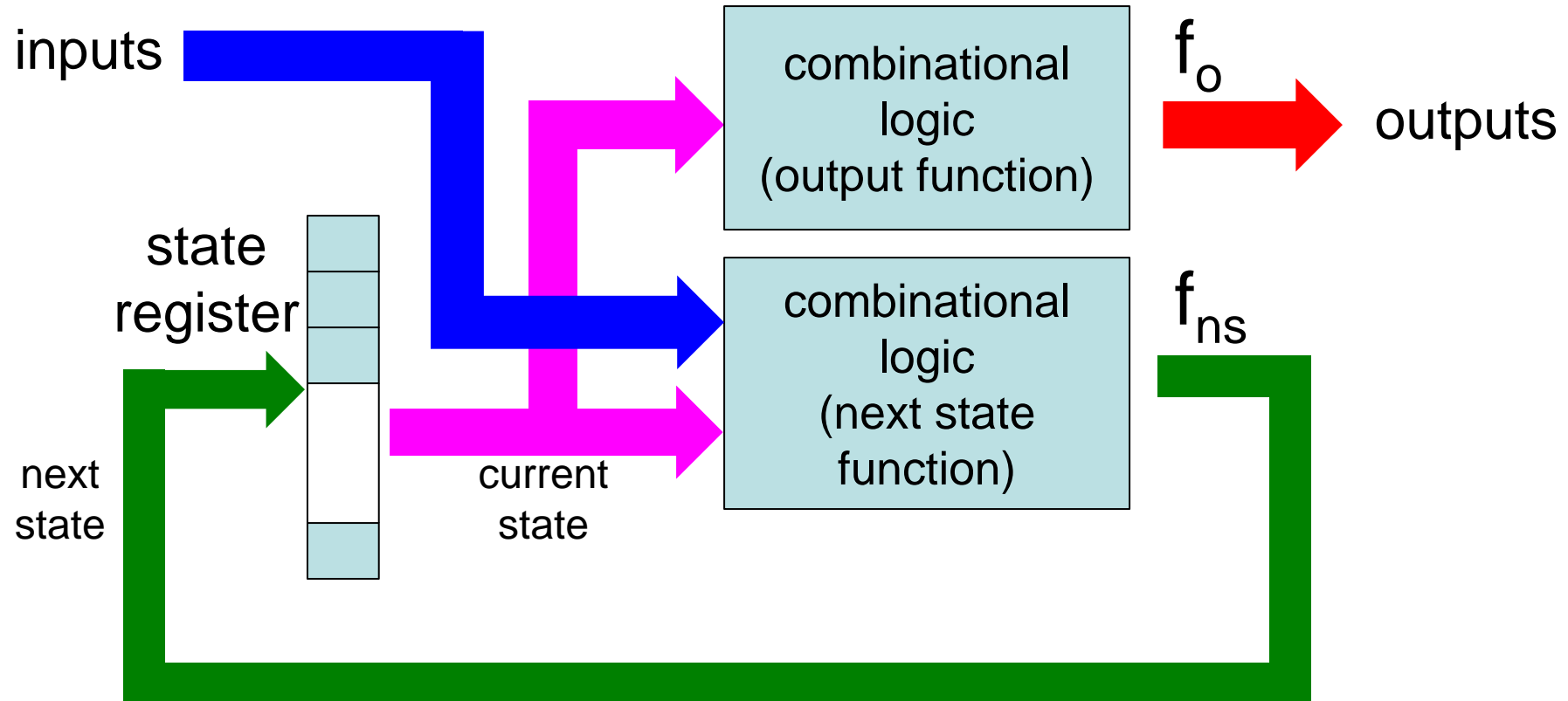
## **Sequential**

Success depends on the **sequence** of values (e.g, R-13, L-22, R-3).

- Combinational: no internal memory
- Sequential requires “remembering” previous input values; the memory is internal to the lock

# Sequential Circuits

(aka Finite State Machines [FSM])



Two key functions must be determined

- Next state function:  $f_{ns}$  (current state, input)
- Output function:  $f_o$  (current state)

# Finite State Machine Design

- The behavior of combinational logic is specified by a truth table
- The behavior of a finite state machine is specified by a state diagram
- Finite state machine design procedure
  - Develop state diagram
  - Determine next state function (comb. logic)
  - Determine output function (comb. logic)



# Example: Candy Machine

Design controller for a vending machine

- Candy costs \$0.15
- Input signals
  - N: asserted (is '1') if \$0.05 coin inserted
  - D: asserted if \$0.10 coin inserted
  - At most one input asserted at a time
  - N/D ignored while candy being dispensed
- Output signal
  - C: asserted for 1 clock cycle to dispense candy
- No other coins are accepted
- Machine gives no change!

# State Diagram

- Try to put together the pieces for a state diagram for this setup.
  - What are the states for the system?
  - What are the transitions that can exist among the states?
- Hint: aim for 4 states and 9 transitions. Note that a transition can include staying in the same state!

Design controller for a vending machine

- Candy costs \$0.15
- Input signals
  - N: asserted (is '1') if \$0.05 coin inserted
  - D: asserted if \$0.10 coin inserted
  - At most one input asserted at a time
  - N/D ignored while candy being dispensed
- Output signal
  - C: asserted for 1 clock cycle to dispense candy
- No other coins are accepted
- Machine gives no change!

# State Diagram

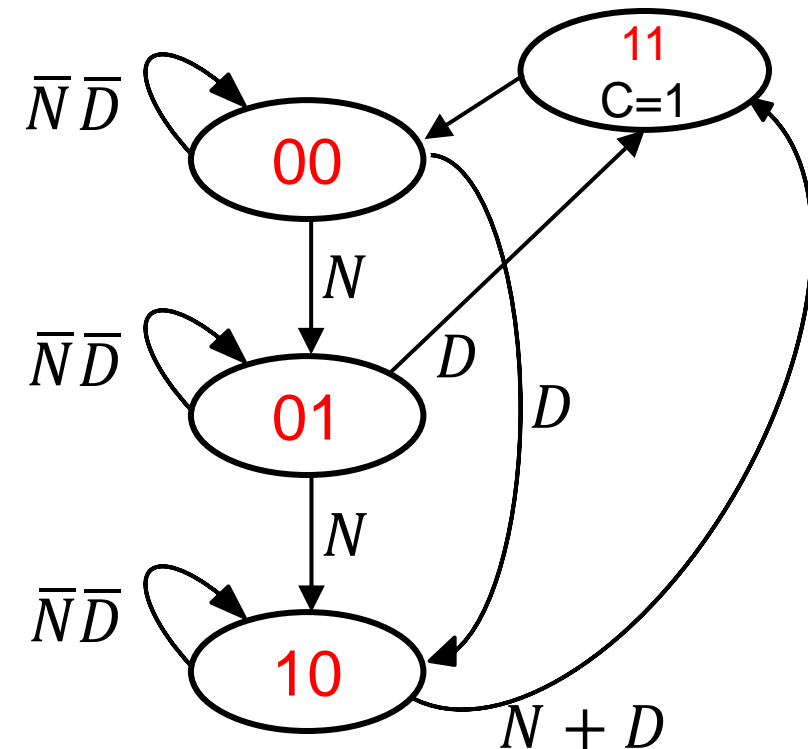
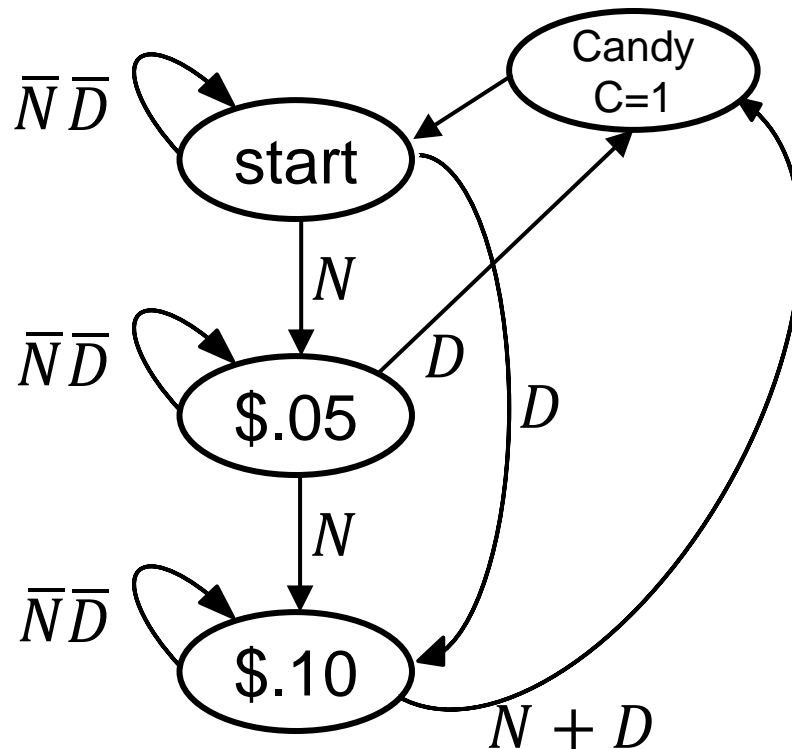
- What are the states for the system?
  - 0 cents accepted
  - 5 cents accepted
  - 10 cents accepted
  - $\geq 15$  cents  $\rightarrow$  triggers candy release
- What are the transitions that can exist among the states?
  - From 0 cents, to 5 or 10 cents.
  - From 5 cents, to 10 or 15 cents/candy.
  - From 10 cents, to 15 cents/candy.
  - From 15 cents/candy, to 0 cents.
  - From 0/5/10 cents, can stay put.

Design controller for a vending machine

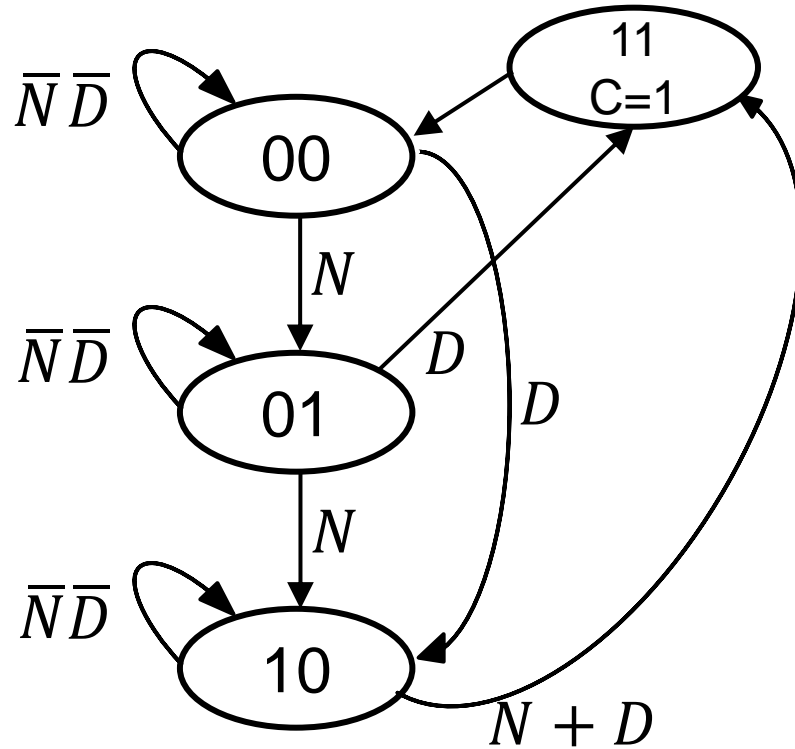
- Candy costs \$0.15
- Input signals
  - N: asserted (is '1') if \$0.05 coin inserted
  - D: asserted if \$0.10 coin inserted
  - At most one input asserted at a time
  - N/D ignored while candy being dispensed
- Output signal
  - C: asserted for 1 clock cycle to dispense candy
- No other coins are accepted
- Machine gives no change!

# State Diagram

- Circle indicates a state
  - Output specified in state
- Arc indicates state transition
- Arc label: <input>



# Next State and Output Functions



$$\begin{aligned}
 q_1 &= \overline{Q_1} \overline{Q_0} D \overline{N} + \overline{Q_1} \overline{Q_0} \overline{D} N + \overline{Q_1} Q_0 D \overline{N} \\
 &\quad + Q_1 \overline{Q_0} \overline{D} \overline{N} + Q_1 \overline{Q_0} \overline{D} N + Q_1 \overline{Q_0} D \overline{N} \\
 q_0 &= \overline{Q_1} \overline{Q_0} \overline{D} N + \overline{Q_1} \overline{Q_0} \overline{D} \overline{N} + \overline{Q_1} Q_0 D \overline{N} \\
 &\quad + Q_1 \overline{Q_0} \overline{D} N + Q_1 \overline{Q_0} D \overline{N} \\
 C &= Q_1 Q_0
 \end{aligned}$$

Current State =  $Q_1 Q_0$

Next State =  $q_1 q_0$

$Q_1$	$Q_0$	$D$	$N$	$q_1$	$q_0$	$C$
0	0	0	0	0	0	0
0	0	0	1	0	1	0
0	0	1	0	1	0	0
0	1	0	0	0	1	0
0	1	0	1	1	0	0
0	1	1	0	1	1	0
1	0	0	0	1	0	0
1	0	0	1	1	1	0
1	0	1	0	1	1	0
1	1	0	0	0	0	1
1	1	0	1	0	0	1
1	1	1	0	0	0	1

One coin at a time: D and N cannot both be 1, so 12 states, not 16

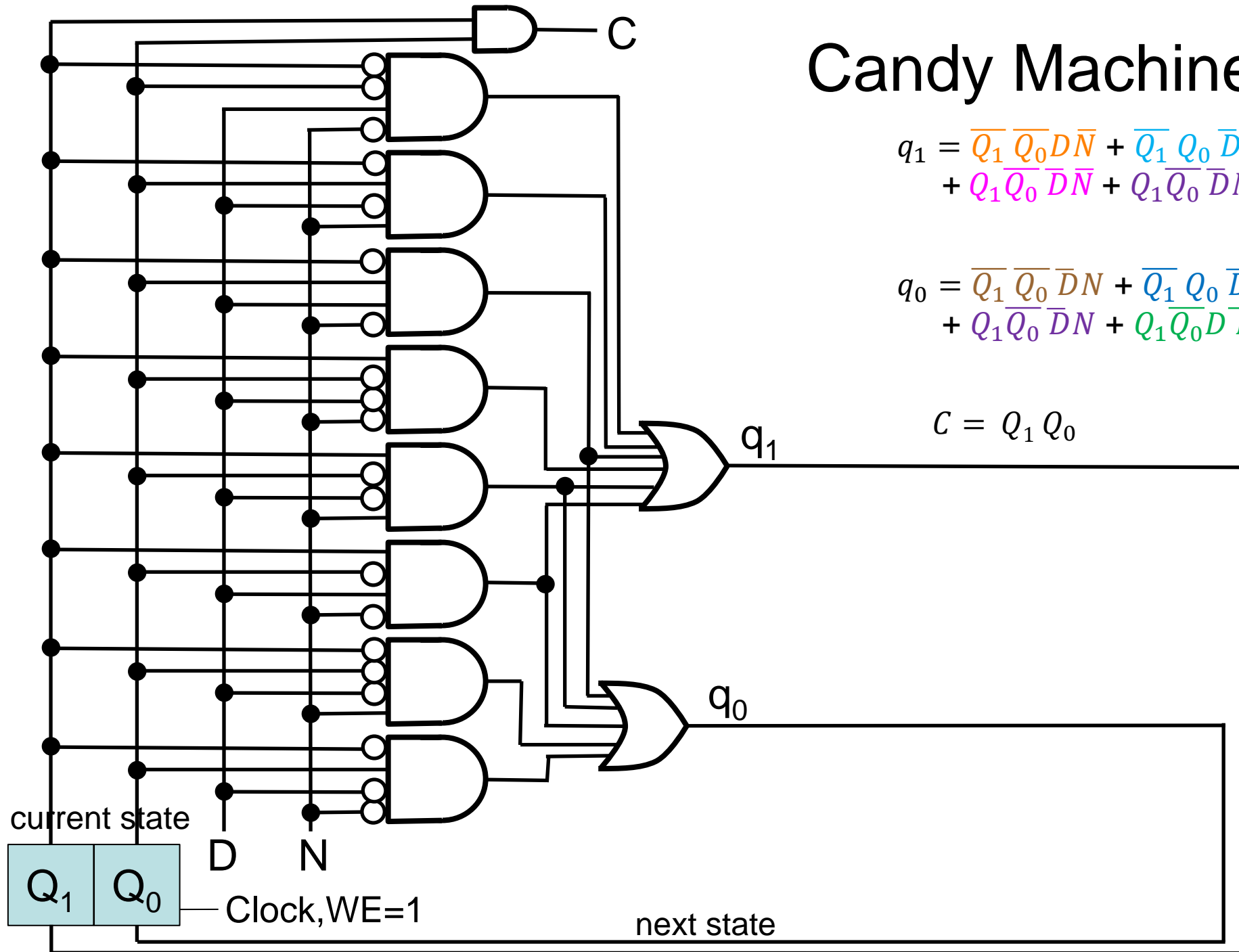
Start from 0¢

Start from 5¢

Start from 10¢

Start from 15¢

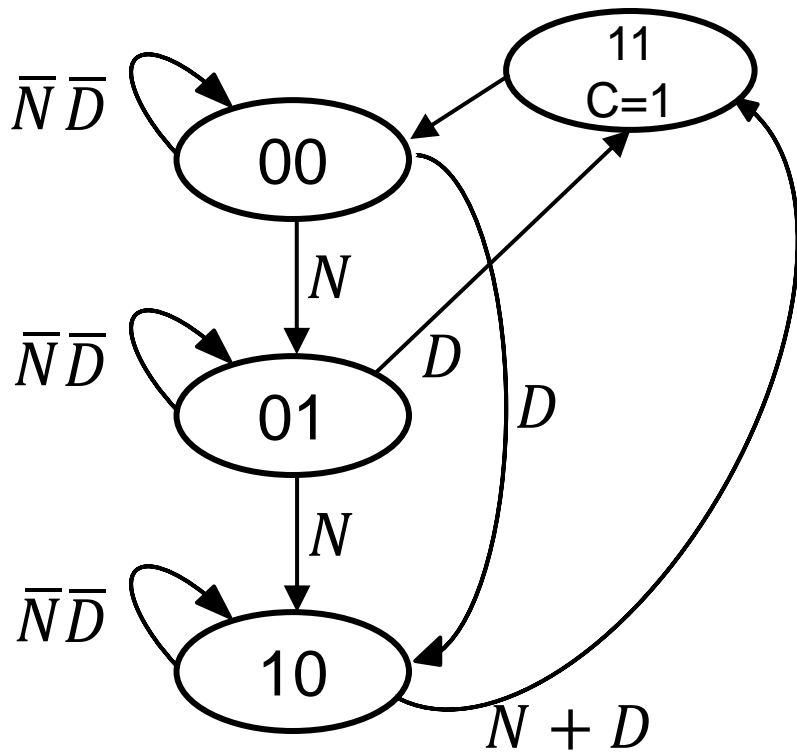
# Candy Machine Circuit



$$q_1 = \overline{Q_1} \overline{Q_0} D \overline{N} + \overline{Q_1} Q_0 \overline{D} N + \overline{Q_1} Q_0 D \overline{N} + Q_1 \overline{Q_0} \overline{D} N + Q_1 \overline{Q_0} \overline{D} N + Q_1 \overline{Q_0} D \overline{N}$$

$$q_0 = \overline{Q_1} \overline{Q_0} \overline{D} N + \overline{Q_1} Q_0 \overline{D} \overline{N} + \overline{Q_1} Q_0 D \overline{N} + Q_1 \overline{Q_0} \overline{D} N + Q_1 \overline{Q_0} D \overline{N}$$

# Time to Complete FSM Operations



State diagram can be used to determine time to perform operations

A state transition arc is followed each clock cycle  
Example (assume 1 Hz clock)

- State 00: 4 clocks waiting\*, then N is asserted (5 clock cycles)
- State 01: 3 clocks waiting\*, then D is asserted (4 clock cycles)
- State 11: 1 clock cycle
- Total: 10 clock cycles or 10 seconds

\* 3 and 4 are arbitrary values used for this example

# Summary

- Memory
  - Organized as a 2D array of 1-bit cells
  - Well suited for fixed-sized data (e.g., 32 bits)
- Sequential circuits needed whenever output depends on prior inputs (memory)
- Sequential circuit design
  - Construct state diagram
  - Design next state function (combinational logic)
  - Design output function (combinational logic)
  - Put it all together (state register, next state function, output function)