

**LOGIC CIRCUITS**

- To build a computer, we must have:
  - ▷ A hardware representation of binary data
    - Bits = signal levels!
    - High or low voltages are mapped to 0 or 1
  - ▷ Ways to do arithmetic
    - Binary arithmetic = logic!
    - The additive inverse is a logic function (bitwise negation) followed by addition:  $-m_{\text{two}} = \bar{m} + 1$
    - A 1-bit adder can be designed from Boolean logic equations:
$$s = a \cdot \bar{b} \cdot \bar{c}_i + \bar{a} \cdot b \cdot \bar{c}_i + \bar{a} \cdot \bar{b} \cdot c_i + a \cdot b \cdot c_i$$
$$c_o = a \cdot b + a \cdot c_i + b \cdot c_i$$
  - To multiply, we just shift and add

## LOGIC CIRCUITS (1)

- Inputs and outputs are “high” or “low” voltages
- Building blocks:
  - ▷ Combinational logic circuits
    - Output depends only on input
    - No feedback or memory
    - Implement truth tables
    - Examples: AND, OR, NAND, NOR, XOR, XNOR gates
  - ▷ Sequential logic circuits
    - Have internal state (memory elements or feedback loops)
    - Normally controlled by a clock signal
    - Examples: Flip-flops, registers, DRAM cells

## SYNCHRONOUS vs. ASYNCHRONOUS LOGIC

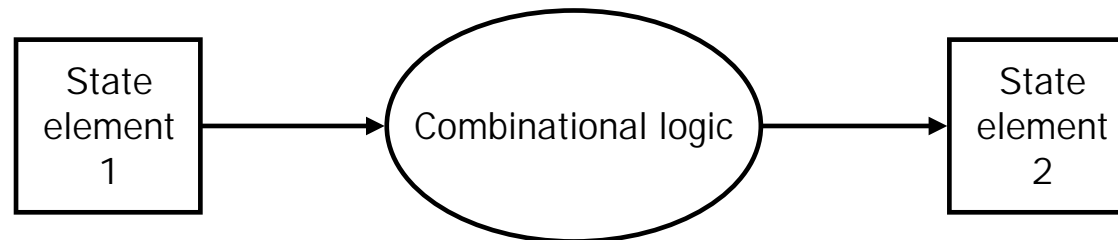
- Asynchronous logic:
  - ▷ No single master clock signal
  - ▷ Examples:
    - Flip-flops, latches
    - Buses (clock frequency different from CPU's clock frequency)
  - ▷ In principle, may be faster than synchronous logic
- Synchronous logic:
  - ▷ Controlled by a master clock signal
  - ▷ State changes are correlated & occur at known times
  - ▷ Easier to design & more reliable than asynchronous logic

## LIMITS ON CLOCK FREQUENCY (1)

- A **clock** is a free-running signal with a fixed frequency

$$\nu_c = \frac{1}{T_c}, \quad T_c = \text{clock period}$$

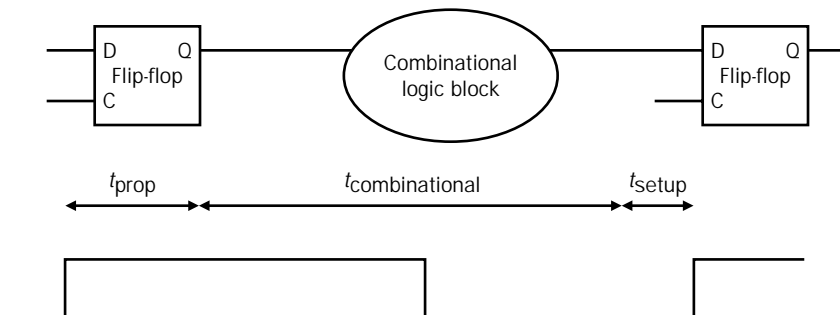
Example:  $T_c = 2.5 \text{ ns}$ ,  $\nu_c = 500 \text{ MHz}$



**LIMITS ON CLOCK FREQUENCY (2)**

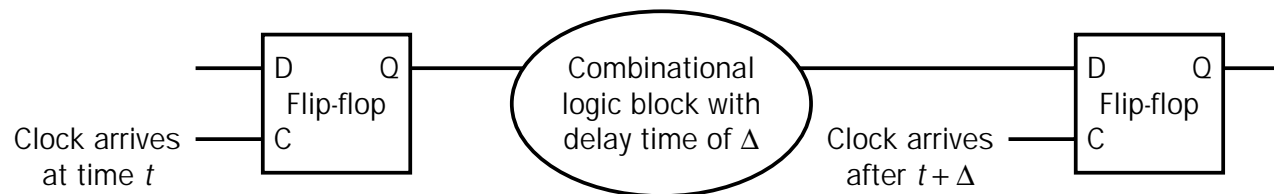
- Delays that determine the minimum clock period (maximum clock frequency):
  - ▷ Transport delay through sequential logic blocks,  $\Delta_t$
  - ▷ Time for signals to settle in combinational logic blocks,  $\Delta_{comb}$
  - ▷ Inertial delay (setup time),  $\Delta_{su}$
  - ▷ Clock skew,  $\Delta_c$
  - ▷ Clock period  $T_c$  must satisfy

$$T_c > \Delta_t + \Delta_{comb} + \Delta_{su} + \Delta_c$$



## LIMITS ON CLOCK FREQUENCY (3)

- Clock skew
  - ▷ Finite signal propagation velocity ( $< c$ ) and different lengths of clock traces  $\Rightarrow$  clock edge arrives at different combinational logic blocks at different times
  - ▷ Can cause a race condition in which a combinational block changes the inputs to the next sequential block before the clock edge arrives



## LIMITS ON CLOCK FREQUENCY (4)

- Heat dissipation
  - ▷ On chip
    - Danger: Thermal runaway
    - Example: Emitter-coupled logic (ECL) is faster than CMOS logic, but dissipates much more heat
    - Smaller feature size
      - ⇒ lower operating voltage ⇒ less heat dissipation
  - ▷ On board
    - Example: Failure of beta Cray Y-MP floating-point boards under heavy use
- Cost
  - ▷ The fastest parts are usually the scarcest & most expensive

## LOGIC GATES (1)


- A **logic gate** is an electronic circuit that implements one of the basic logical functions (AND, OR, NOT)

▷ Truth tables:

$a$	$\bar{a}$
0	1
1	0

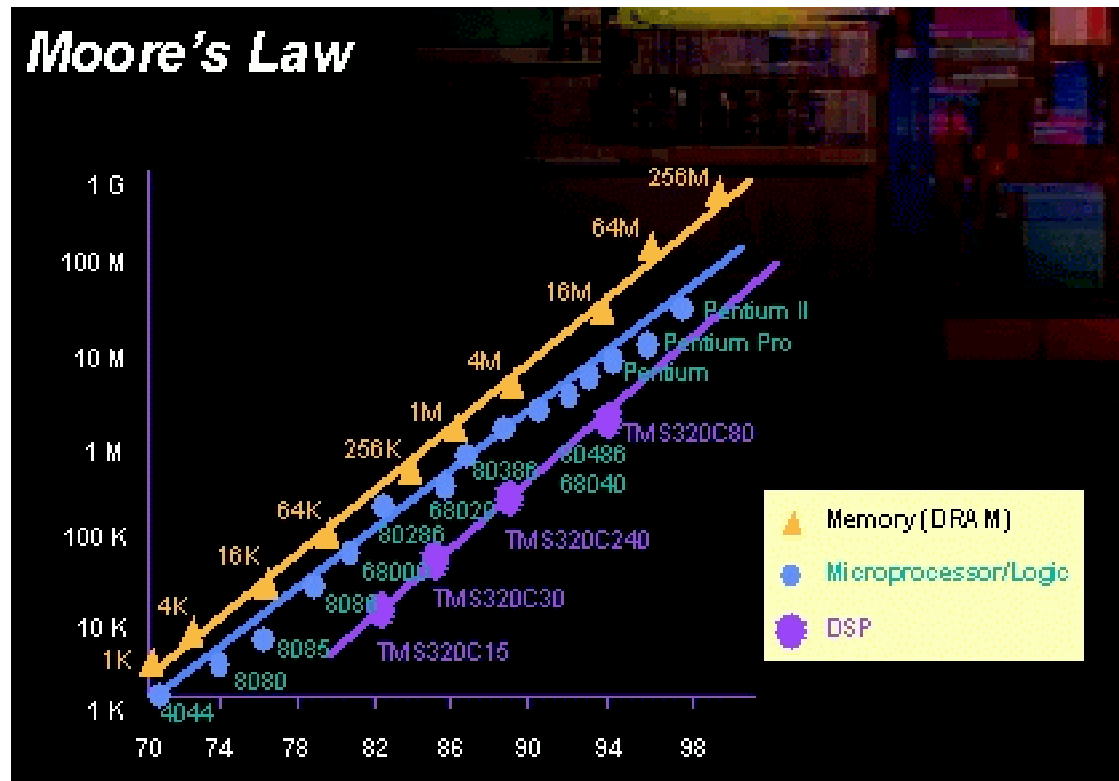
$a$	$b$	$a + b$	$a \cdot b$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

## LOGIC GATES (2)

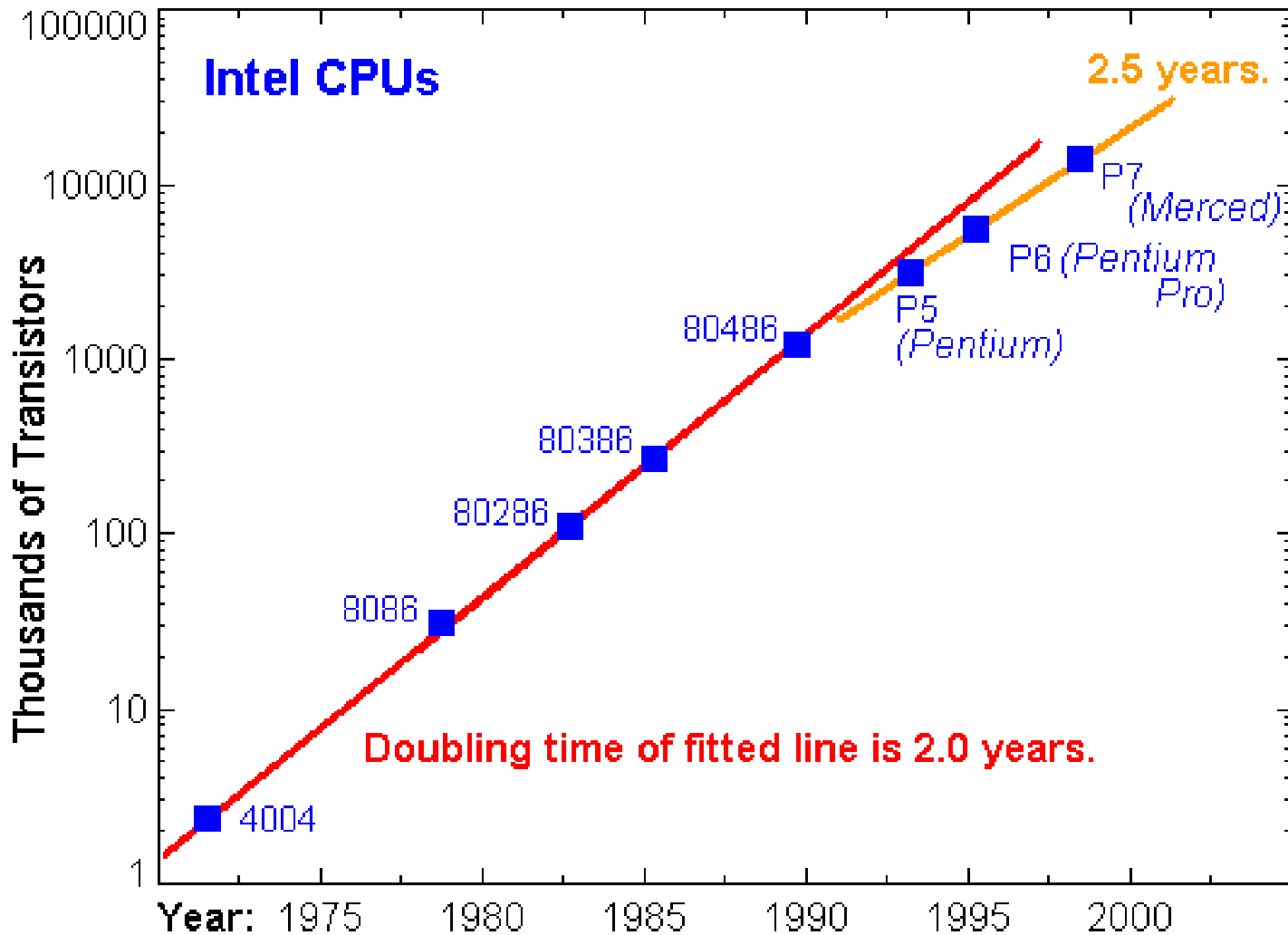
- AND, OR, and NOT can be implemented using
  - ▷ Mechanical devices
  - ▷ Mechanical switches 
  - ▷ Electromechanical switches
  - ▷ Transistors
- Why did transistors win out?
  - ▷ Moore's "law": The number of transistors on a microprocessor die doubles, and the cost per transistor halves, every 18–24 months
    - Dr. Gordon Moore is the founder of Intel



# MOORE'S LAW (1)



# MOORE'S LAW (2)



**LOGICAL “AND”**

- The **AND** function has the value “true” when both of its inputs (arguments) are “true”
  - ▷ Two arguments (inputs)  $a, b$
  - ▷ One output:

$a$	$b$	$a$ AND $b$
0	0	0
0	1	0
1	0	0
1	1	1

- ▷ Boolean notation:

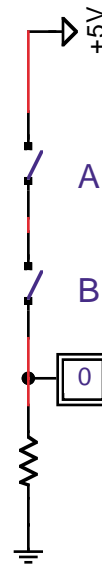
$$\text{AND}(a, b) = a \cdot b$$

A logical product, equal to the numerical product of two bits

- ▷ An **and** gate implements the AND function: 

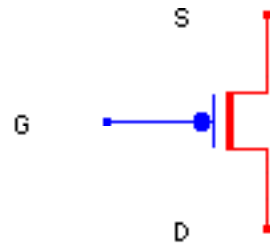
# IMPLEMENTING “AND” USING SWITCHES

A AND B

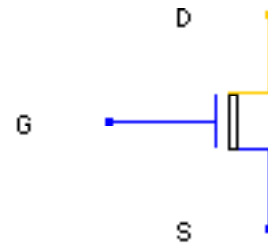


# TRANSISTORS AS SWITCHES (1)

P-type transistor



N-type transistor



[VCC]

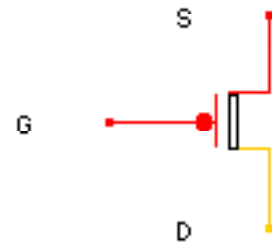
[GND]

[Z (floating)]

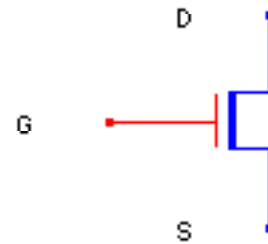
[Short-circuit]

## TRANSISTORS AS SWITCHES (2)

P-type transistor



N-type transistor



[VCC]

[GND]

[Z (floating)]

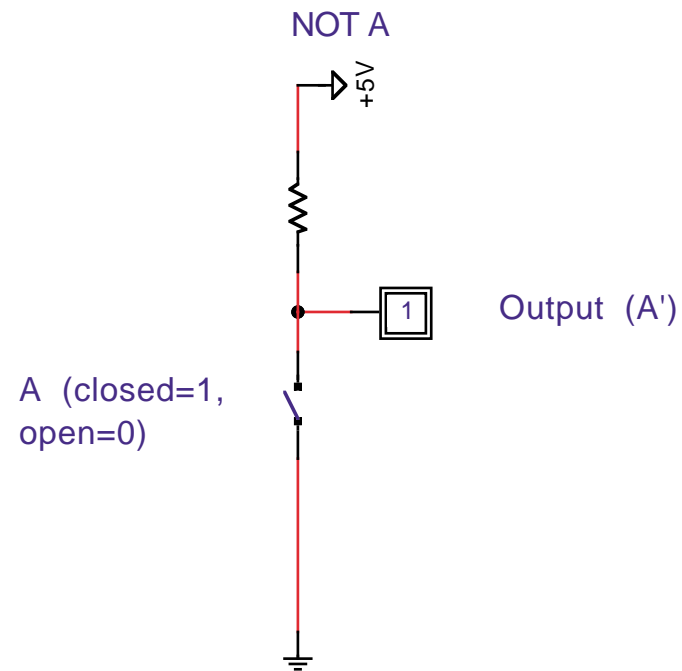
[Short-circuit]

**LOGICAL NEGATION**

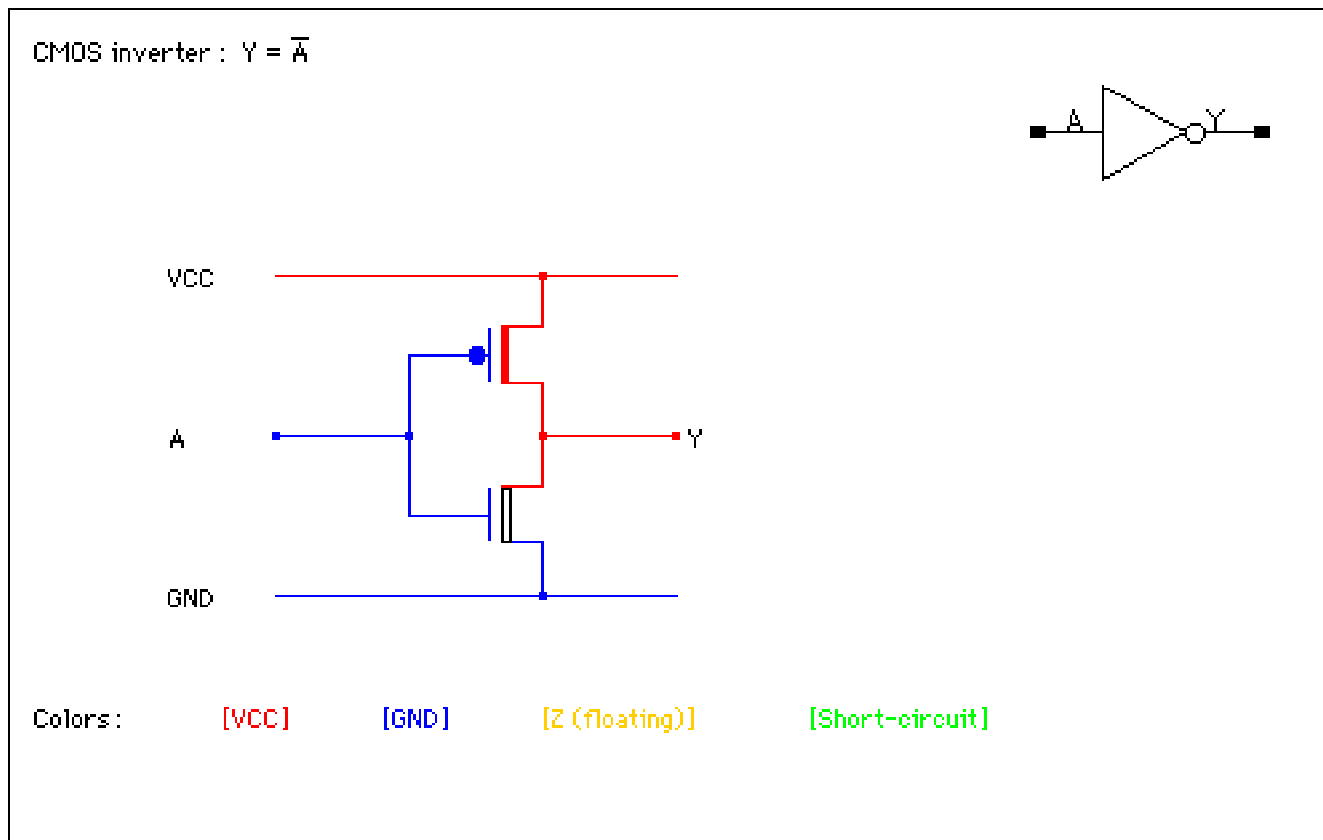
- The operation of **negation** or **inversion** changes the value of a logical variable
  - ▷ The name of the negation operator is NOT
  - ▷ The **complement** of a logical variable  $a$  is the logical variable  $\bar{a}$  such that:
    - $\bar{a}$  is true when  $a$  is false
    - $\bar{a}$  is false when  $a$  is true
  - Then:  $\bar{a} = \text{NOT}(a)$
  - ▷ An **inverter** gate implements 1-bit negation:

$$a \text{ --- } \triangleleft \text{ --- } \text{NOT}(a) = \bar{a}$$

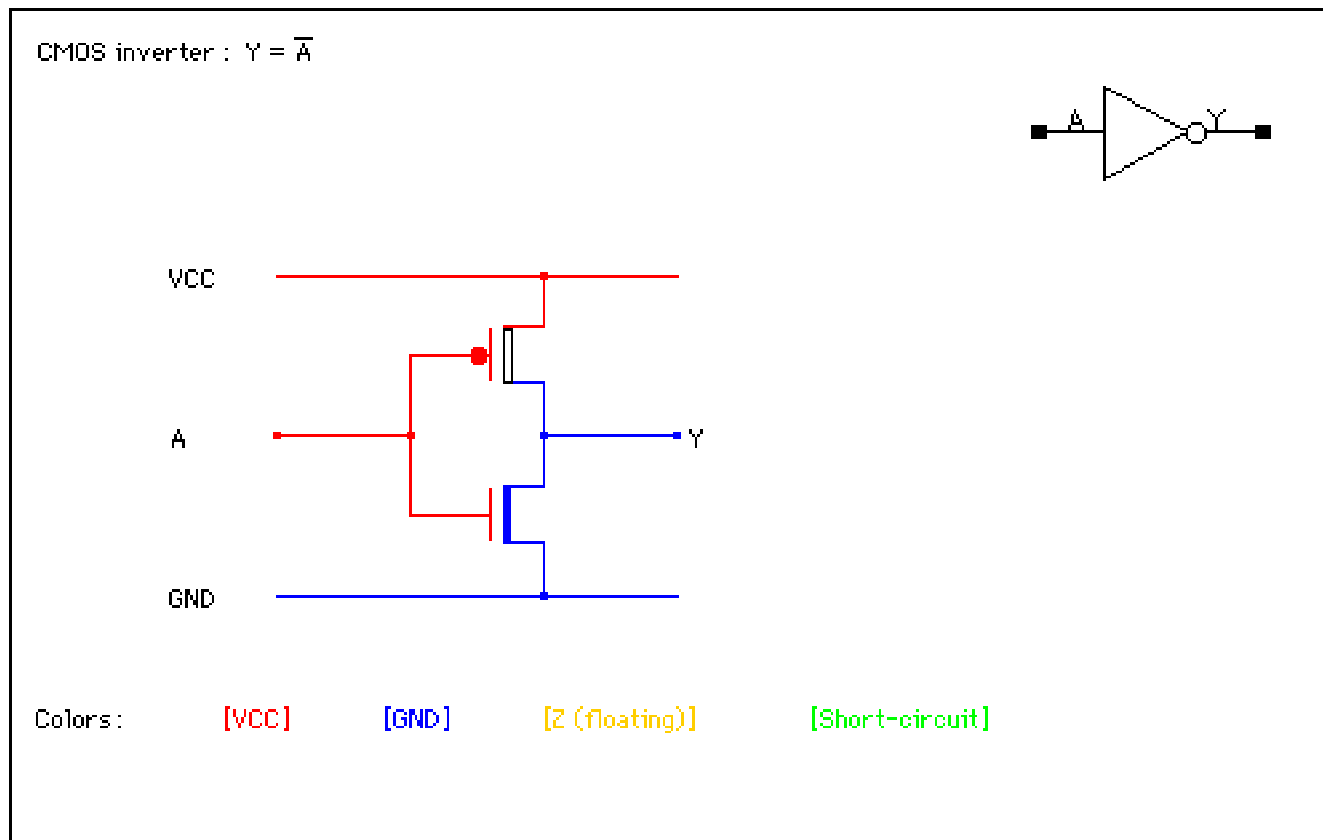
# IMPLEMENTING “NOT” USING SWITCHES



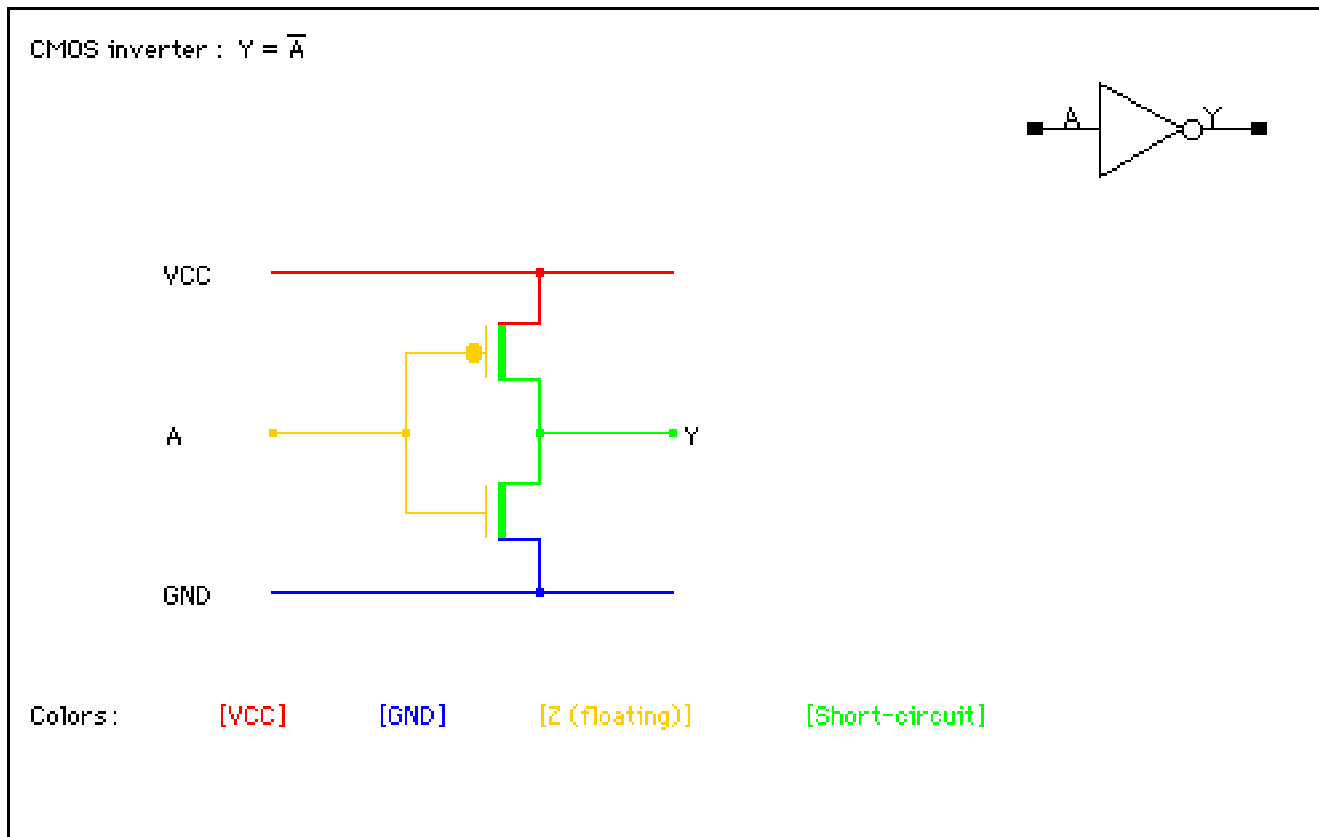
# IMPLEMENTING “NOT” USING TRANSISTORS (1)



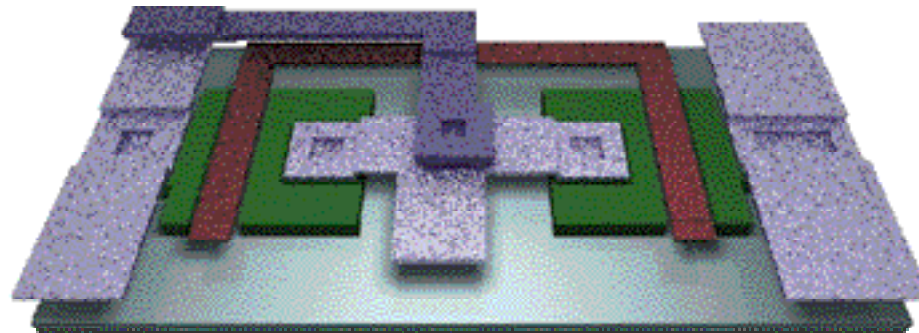
# IMPLEMENTING “NOT” USING TRANSISTORS (2)



# IMPLEMENTING “NOT” USING TRANSISTORS (3)



## IMPLEMENTING “NOT” USING TRANSISTORS (4)



## LOGICAL “NAND”

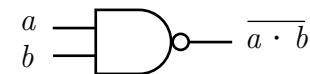
- The **NAND** function has the value “false” when both of its inputs (arguments) are “true”; otherwise its value is “true”
  - ▷ Two arguments (inputs)  $a, b$
  - ▷ One output:

$a$	$b$	$a \text{ NAND } b$
0	0	1
0	1	1
1	0	1
1	1	0

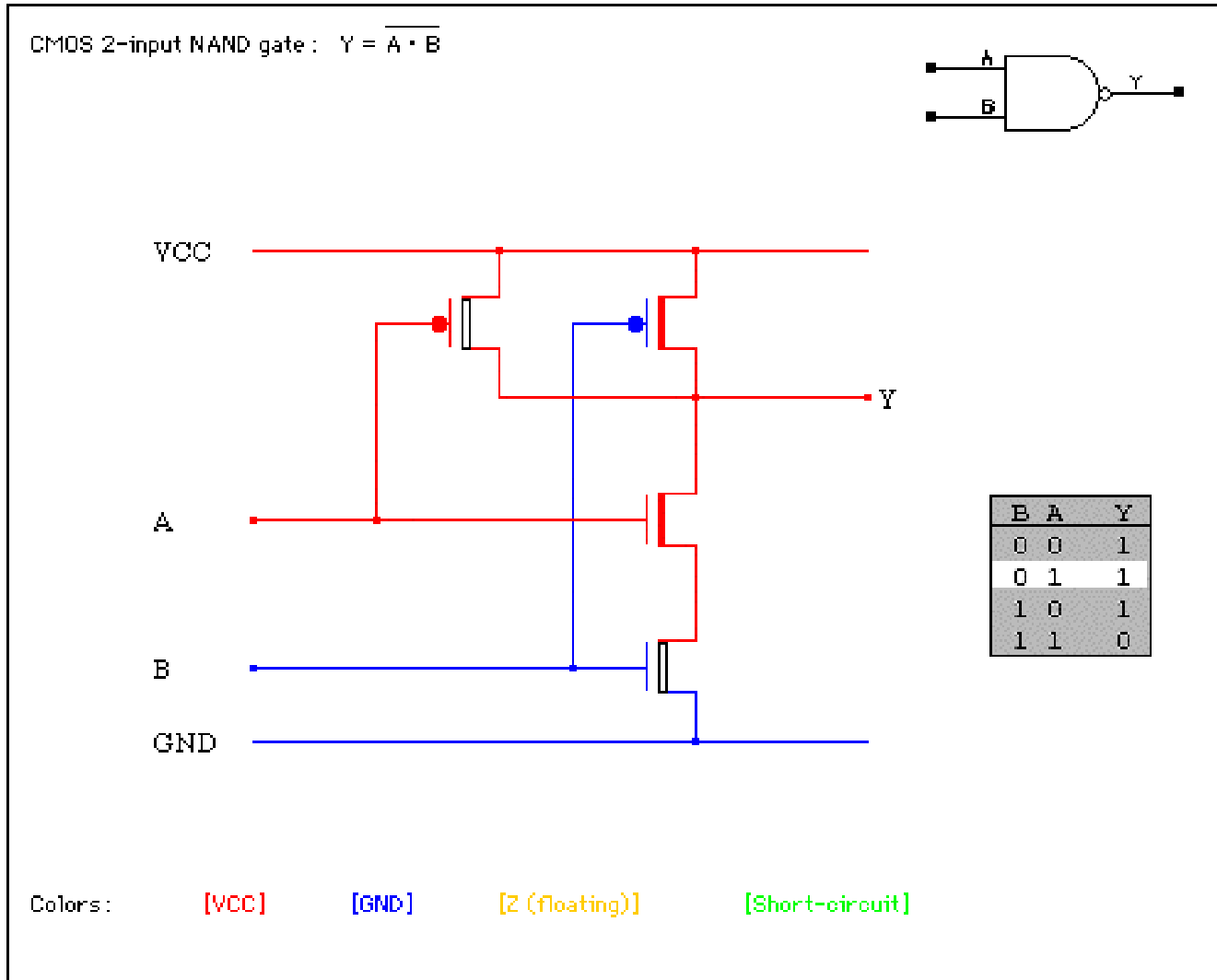
- ▷ Boolean notation:

$$\text{NAND}(a, b) = \overline{a \cdot b}$$

- ▷ A **nand** gate implements the NAND function:



# IMPLEMENTING "NAND" USING TRANSISTORS



## LOGICAL “OR”

- The **OR** function has the value “true” when at least one of its inputs (arguments) is “true”
  - ▷ Two arguments (inputs)  $a, b$
  - ▷ One output:

$a$	$b$	$a$ OR $b$
0	0	0
0	1	1
1	0	1
1	1	1

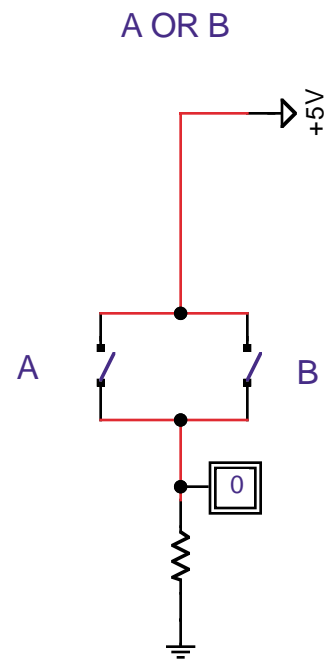
- ▷ Boolean notation:

$$\text{OR}(a, b) = a + b$$

A logical sum, not an arithmetic sum!

- ▷ An **or** gate implements the OR function: 

# IMPLEMENTING “OR” USING SWITCHES



**LOGICAL “NOR”**

- The **NOR** function has the value “false” when at least one of its inputs (arguments) is “true”
  - ▷ Two arguments (inputs)  $a, b$
  - ▷ One output:

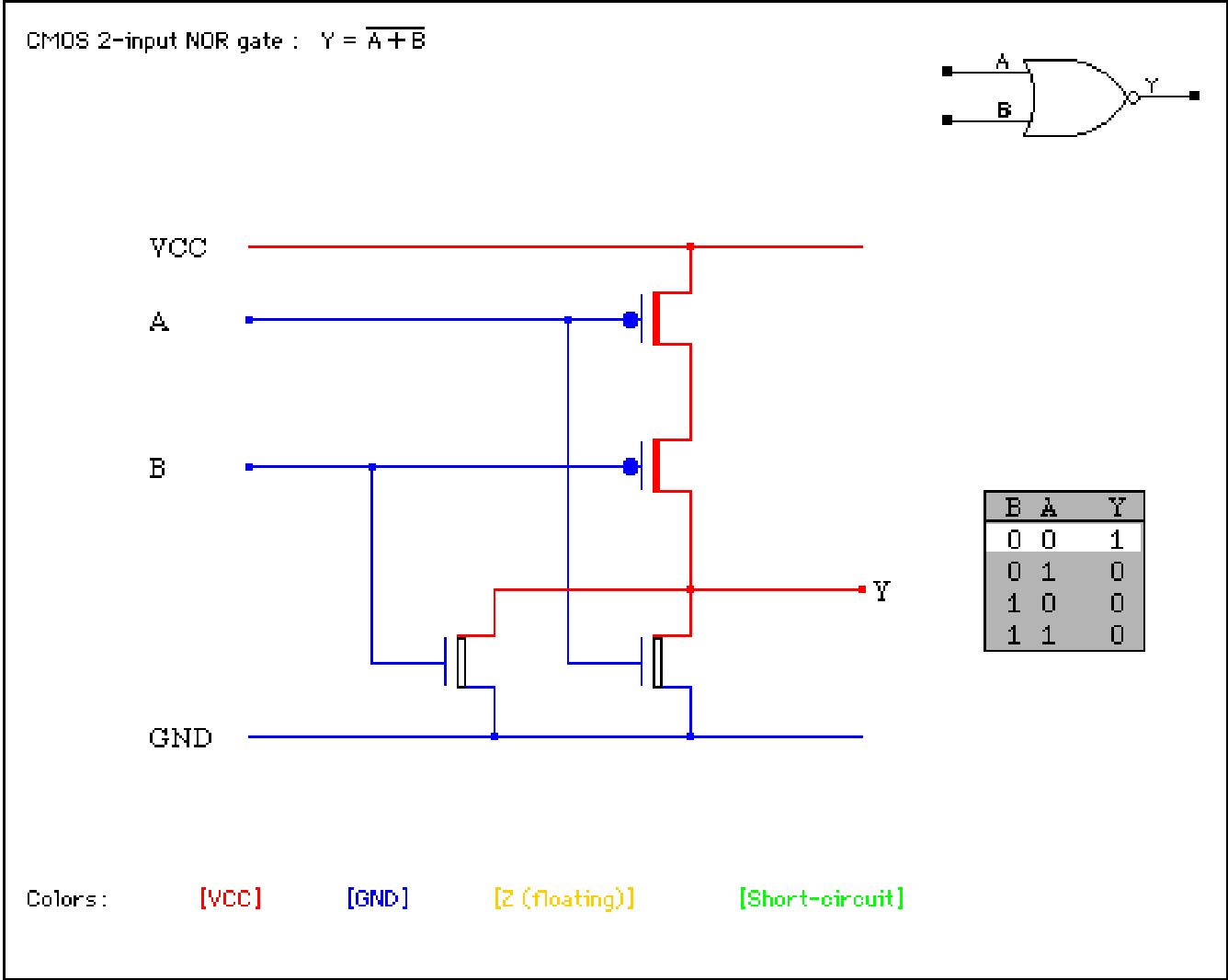
$a$	$b$	$a \text{ NOR } b$
0	0	1
0	1	0
1	0	0
1	1	0

- ▷ Boolean notation:

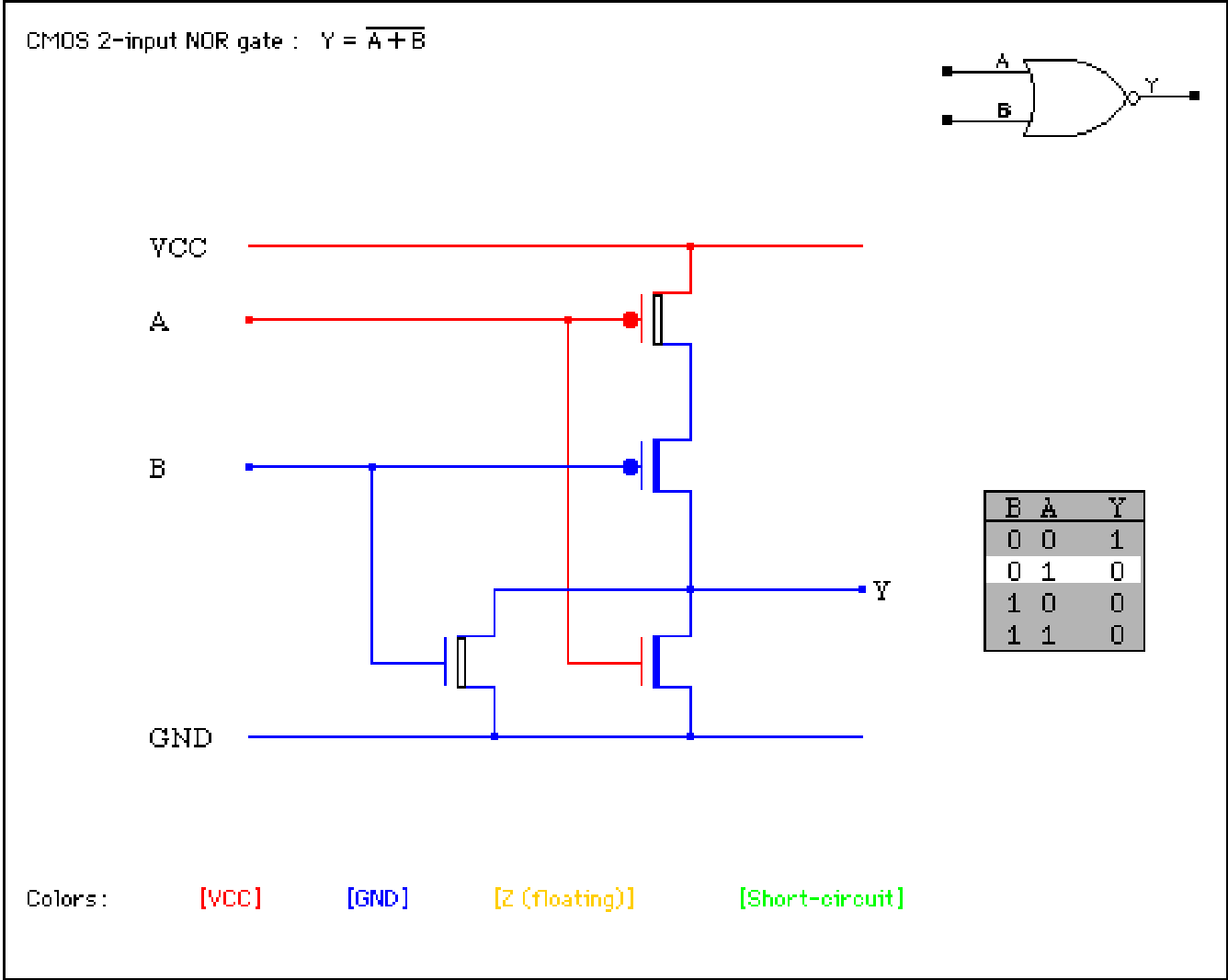
$$\text{NOR}(a, b) = \overline{a + b} \quad (= \text{NOT}(a + b))$$

- ▷ A **nor** gate implements the NOR function: 

# IMPLEMENTING “NOR” USING TRANSISTORS (1)



# IMPLEMENTING “NOR” USING TRANSISTORS (2)



**LOGICAL “XOR”**

- The **XOR** (exclusive OR) function has the value “true” when one and only one of its inputs (arguments) is “true”
  - ▷ Two arguments (inputs)  $a, b$
  - ▷ One output:

$a$	$b$	$a \text{ XOR } b$
0	0	0
0	1	1
1	0	1
1	1	0

- ▷ Boolean notation:

$$\text{XOR}(a, b) = a \oplus b$$

- ▷ An **xor** gate implements the XOR function: 


## LOGICAL “XNOR”

- The **XNOR** (or **coincidence**) function has the value “true” when both of its inputs (arguments) are the same
  - ▷ Two arguments (inputs)  $a, b$
  - ▷ One output:

$a$	$b$	$a \text{ XNOR } b$
0	0	1
0	1	0
1	0	0
1	1	1

- ▷ Boolean notation:

$$\text{XNOR}(a, b) = a \odot b$$

- ▷ An **xnor** gate implements the XNOR function: 

## LOGIC CIRCUITS (2)

- Important parameters of a logic circuit:
  - ▷ Cost
    - Proportional to number of gates
      - ◇ Exclude NOTs (they're cheap)
  - ▷ Power dissipation
    - Average power  $\propto C_{av}V_{DD}^2fN_g$  where
      - ◇  $N_g$  = number of gates,  $C_{av}$  = average capacitance per gate,  
 $V_{DD}$  = supply voltage,  $f$  = frequency at which gates are switched
  - ▷ Maximum delay
    - Proportional to maximum number of gates through which a signal may pass, going from input to output
    - Imposes a lower limit on clock period
      - ⇒ upper limit on clock frequency
      - ⇒ upper limit on performance

## LOGIC CIRCUITS (3)

- Methods for minimizing gate count:
  - ▷ Karnaugh maps
    - Not really useful for more than 5 logical variables
    - Academic teaching tool
  - ▷ Boolean algebra
    - Industrial solution
    - Performed by software
  - ▷ Quine-McCluskey tabular method
- Worst-case time required to minimize a Boolean expression involving  $n$  variables is proportional to  $2^n$

## LOGIC CIRCUITS (4)

- Every Boolean function can be implemented with only 2 levels of logic
  - ▷ One level of gates is ANDs
  - ▷ The other level is ORs
  - ▷ If each input is not available in both direct and complemented form, a level of inverters may be necessary
  - ▷ Example: **PLA** (programmable logic array)

**LOGIC CIRCUITS (5)**

- Canonical **sum of products (SOP)** form
  - ▷ In each product, every variable appears once & only once (either in direct or in complemented form) (**minterms**)
    - 2 levels of logic  $\Rightarrow$  minimal delay
    - If both direct and complemented inputs are not available, may need another level of logic for negation
- Canonical **product of sums (POS)** form
  - ▷ In each sum, every variable appears once & only once (either in direct or in complemented form) (**maxterms**)
  - ▷ Less common than SOP form

**MULTIPLEXORS (1)**

## ● 2-to-1 multiplexor:

▷ 3 inputs:  $s$ ,  $a$ ,  $b$ ▷ 1 output:  $m$ ○ If  $s = 0$ ,  $m = a$ ○ If  $s = 1$ ,  $m = b$ 

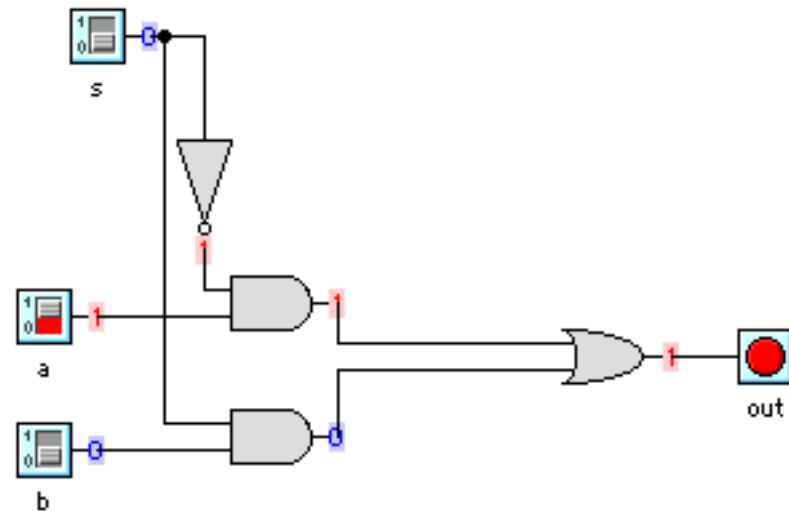
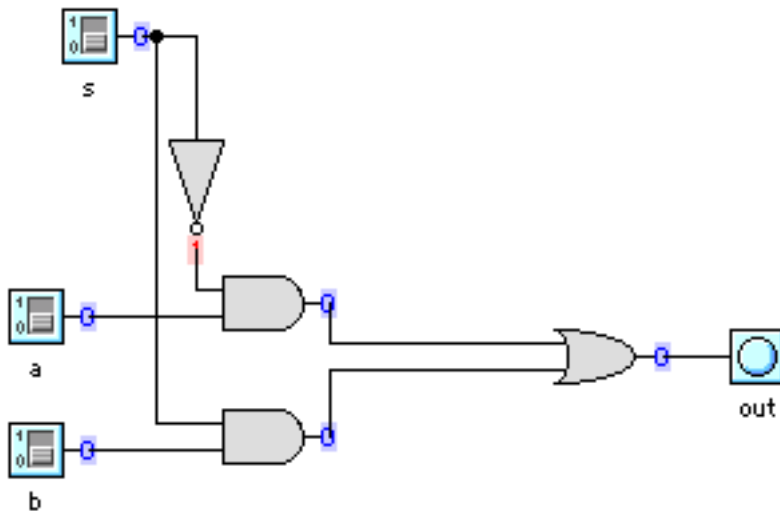
## ● Truth table:

$s$	$a$	$b$	$m$
0	0	don't care	0
0	1	don't care	1
1	don't care	0	0
1	don't care	1	1

● Logic equation:  $m = \bar{s} \cdot a + s \cdot b$

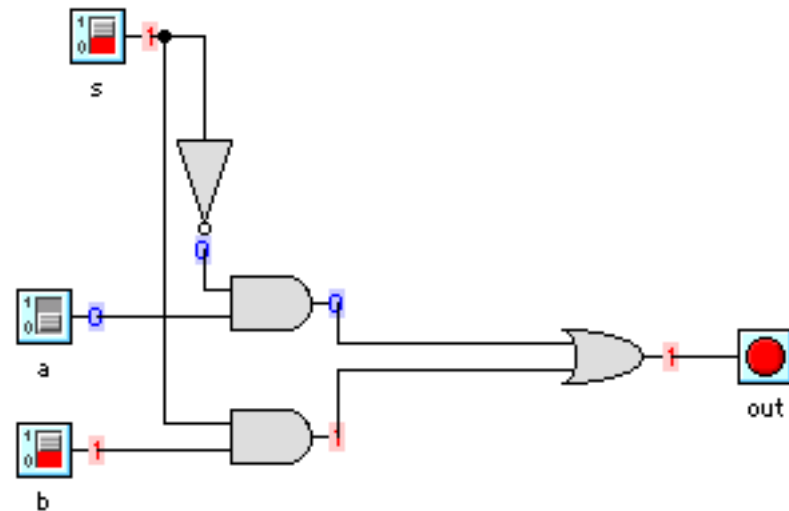
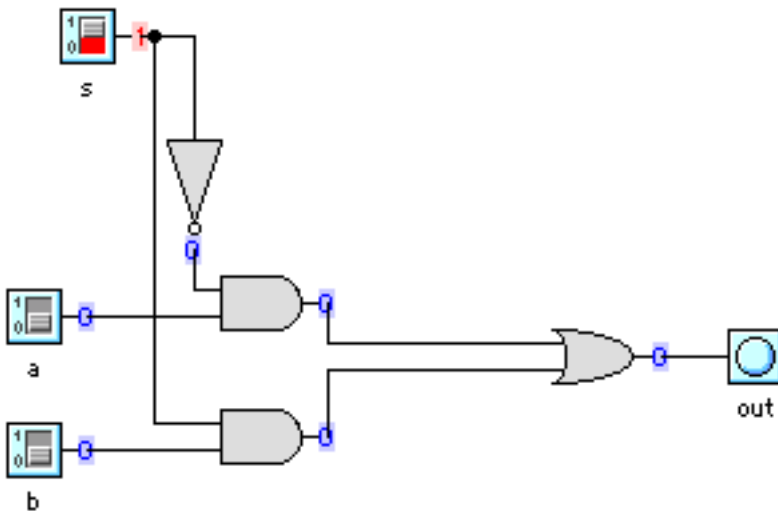
## MULTIPLEXORS (2)

- 2-to-1 multiplexor
  - ▷ Selector input is  $s = 0$
  - ▷ Output is  $m = a$



**MULTIPLEXORS (3)**

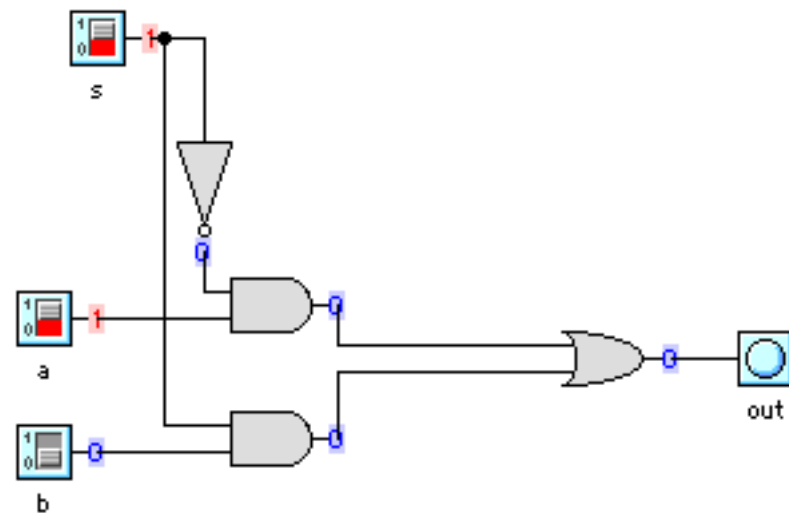
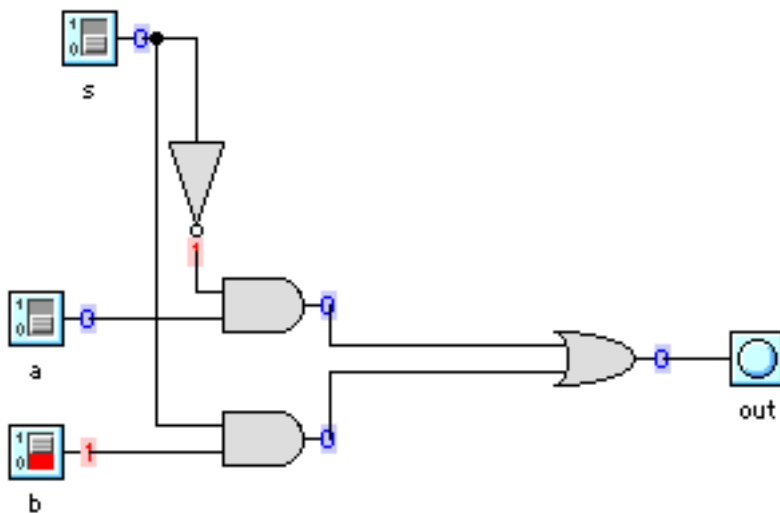
- 2-to-1 multiplexor
  - ▷ Selector input is  $s = 1$
  - ▷ Output is  $m = b$



## MULTIPLEXORS (4)

- 2-to-1 multiplexor

- ▷ If selector input is  $s = 0$ , then  $b$  is blocked
- ▷ If selector input is  $s = 1$ , then  $a$  is blocked



**DECODERS (1)**

- An  $n$ -to- $2^n$  **decoder** is a combinational logic circuit that:
  - ▷ Has  $n$  inputs and  $2^n$  outputs
  - ▷ Converts from a binary representation of a number  $m$  to a logical 1 on output  $m$  and logical 0 on all other outputs ( $0 \leq m \leq 2^n - 1$ )
    - Every string of  $n$  bits corresponds to a unique pattern of logical 1's and 0's on the  $n$  inputs
    - Every string of  $n$  bits is the binary representation of a unique unsigned integer,  $m$
    - Example: 2-bit decoder ( $b_0, b_1 = 1$  (asserted) or 0 (deasserted))

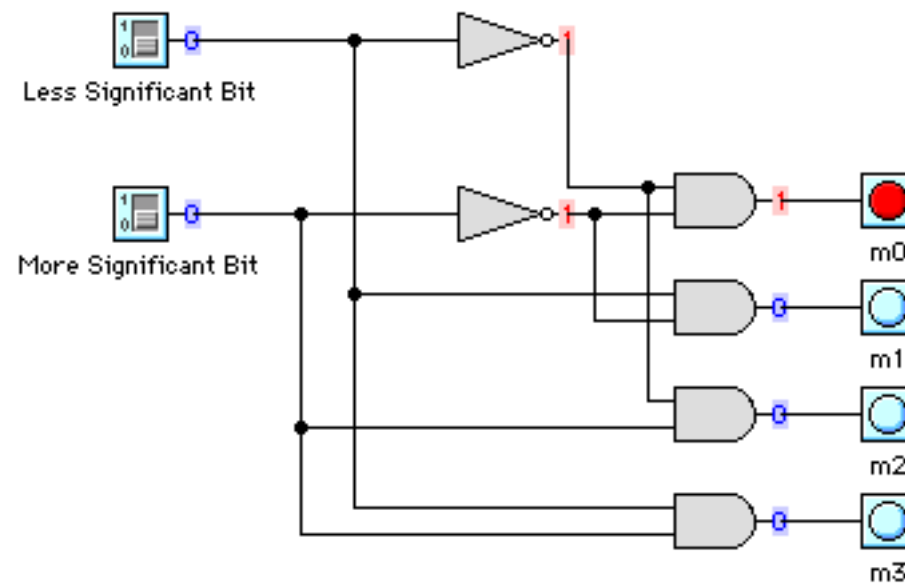
$$m = 2b_1 + b_0 \quad (\text{base 10})$$

- Logic equations:

$$m_0 = \bar{b}_1\bar{b}_0, \quad m_1 = \bar{b}_1b_0, \quad m_2 = b_1\bar{b}_0, \quad m_3 = b_1b_0$$

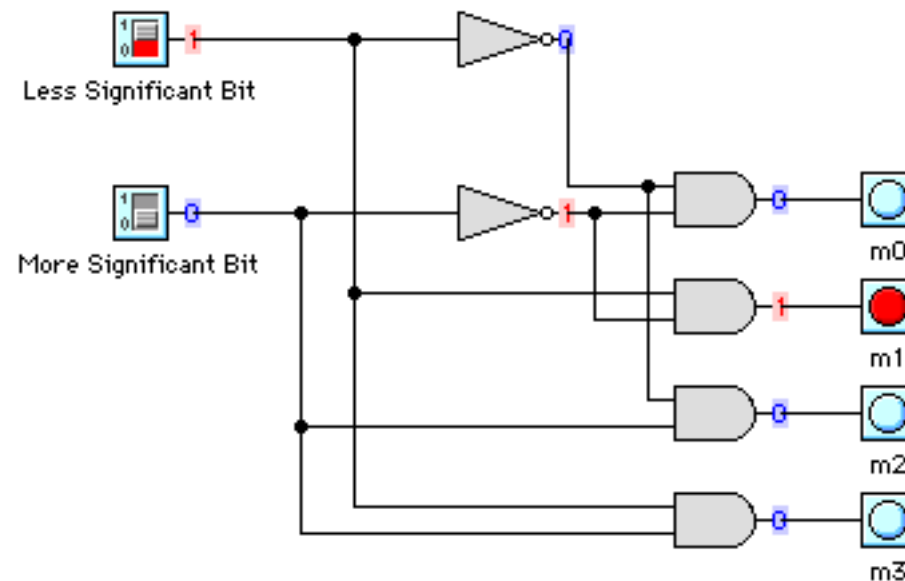
**DECODERS (2)**

## ● 2-bit decoder

▷ Inputs are  $b_0 = 0$ ,  $b_1 = 0$ ▷ Outputs are  $m_0 = 1$ ,  $m_1 = 0$ ,  $m_2 = 0$ ,  $m_3 = 0$ 

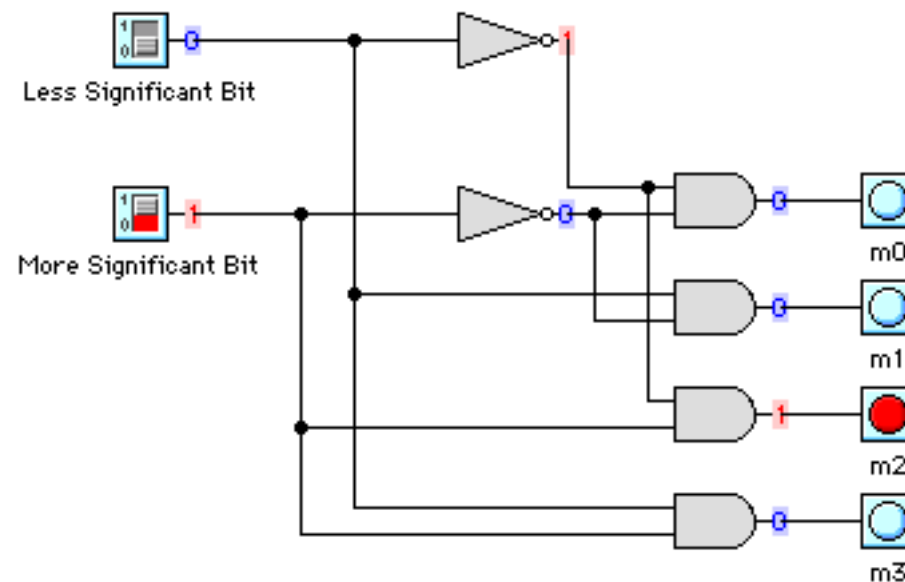
**DECODERS (3)**

## ● 2-bit decoder

▷ Inputs are  $b_0 = 1$ ,  $b_1 = 0$ ▷ Outputs are  $m_0 = 0$ ,  $m_1 = 1$ ,  $m_2 = 0$ ,  $m_3 = 0$ 

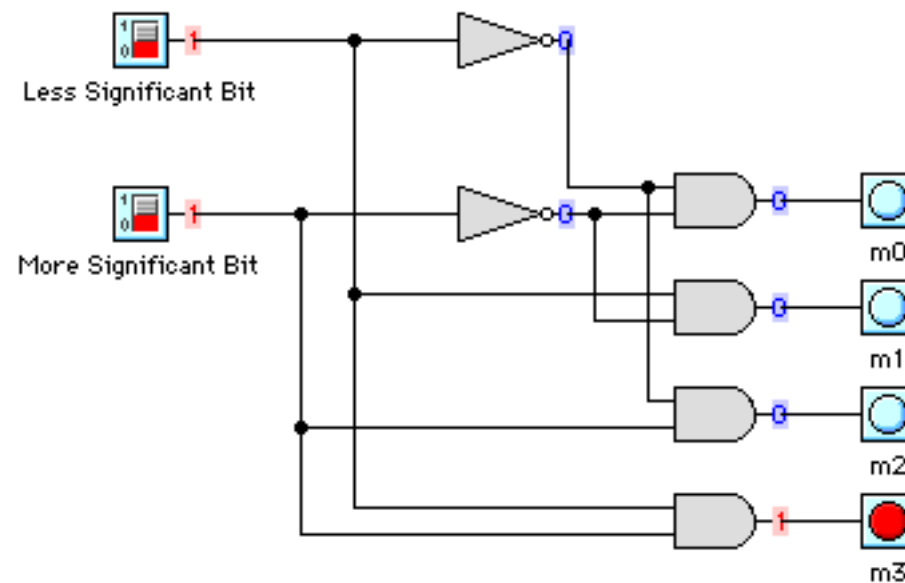
**DECODERS (4)**

## ● 2-bit decoder

▷ Inputs are  $b_0 = 0$ ,  $b_1 = 1$ ▷ Outputs are  $m_0 = 0$ ,  $m_1 = 0$ ,  $m_2 = 1$ ,  $m_3 = 0$ 

**DECODERS (5)**

## ● 2-bit decoder

▷ Inputs are  $b_0 = 1$ ,  $b_1 = 1$ ▷ Outputs are  $m_0 = 0$ ,  $m_1 = 0$ ,  $m_2 = 0$ ,  $m_3 = 1$ 

***n*-BIT DECODER**

- *n*-bit decoder
  - ▷ Address decoding for memory locations, registers or I/O devices
    - Each memory location (or register, or I/O port or device) is assigned a unique *n*-bit address
    - The CPU or I/O controller accesses the target by sending the address over *n* parallel signal lines
    - The decoder activates one of  $2^n$  select lines to access the location or device
    - Example: SCSI-2 bus
      - ◇ Maximum of  $8 = 2^3$  devices, numbered from 0 to 7
      - ◇ The SCSI controller asserts one of 8 lines, depending on the 3-bit address
  - ▷ Can also be used to generate minterms in logic circuits

**TRUTH TABLE FOR 1-BIT HALF ADDER**

- **Half adder:**

- ▷ 2 inputs:  $a$ ,  $b$

- ▷ 2 outputs:  $s$  (Sum),  $c_o$  (CarryOut)

- Truth table:

$a$	$b$	$c_o$	$s$	$c_o$ term	$s$ term
0	0	0	0		
1	0	0	1		$a\bar{b}$
0	1	0	1		$\bar{a}b$
1	1	1	0	$ab$	

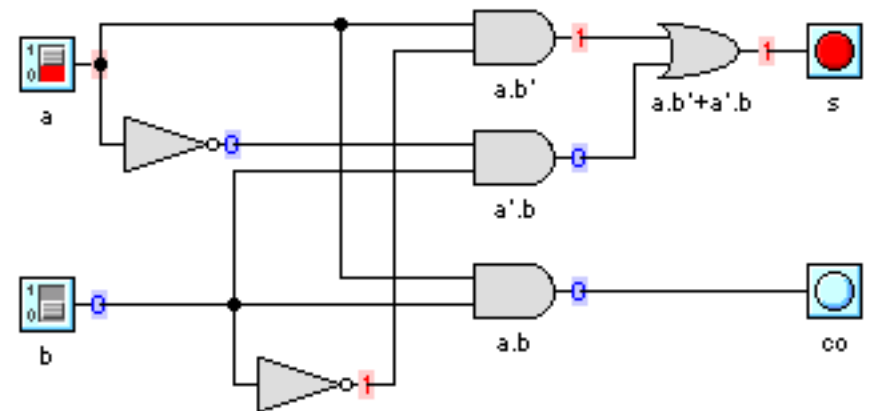
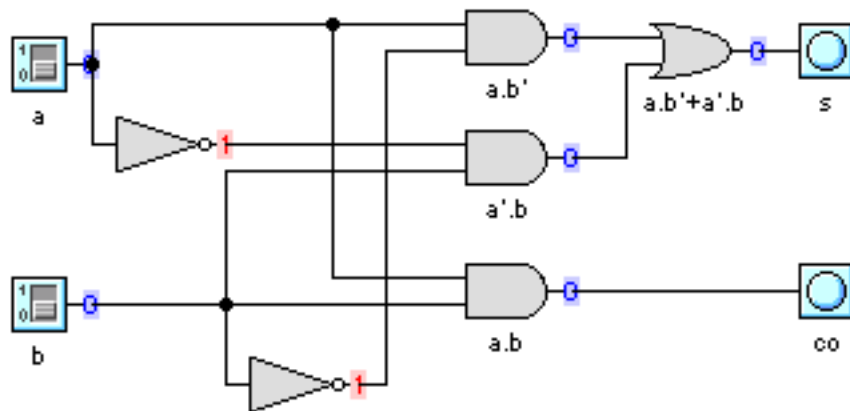
- Logic equations from truth table (rows in which output = 1):

$$s = a \cdot \bar{b} + \bar{a} \cdot b$$

$$c_o = a \cdot b$$

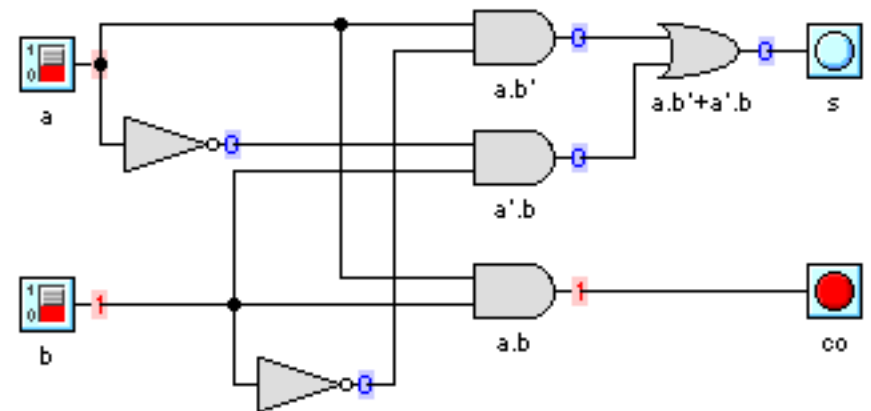
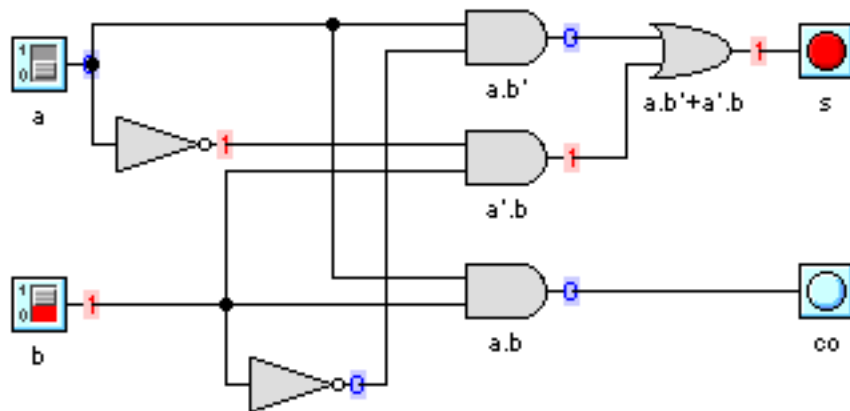
# 1-BIT HALF ADDER (1)

- Inputs on left are  $a = 0, b = 0$
- Inputs on right are  $a = 1, b = 0$



## 1-BIT HALF ADDER (2)

- Inputs on left are  $a = 0, b = 1$
- Inputs on right are  $a = 1, b = 1$



## TRUTH TABLE FOR 1-BIT FULL ADDER

- One-bit full adder:

- ▷ 3 inputs:  $a$ ,  $b$ ,  $c_i$  (CarryIn — from less significant bit)

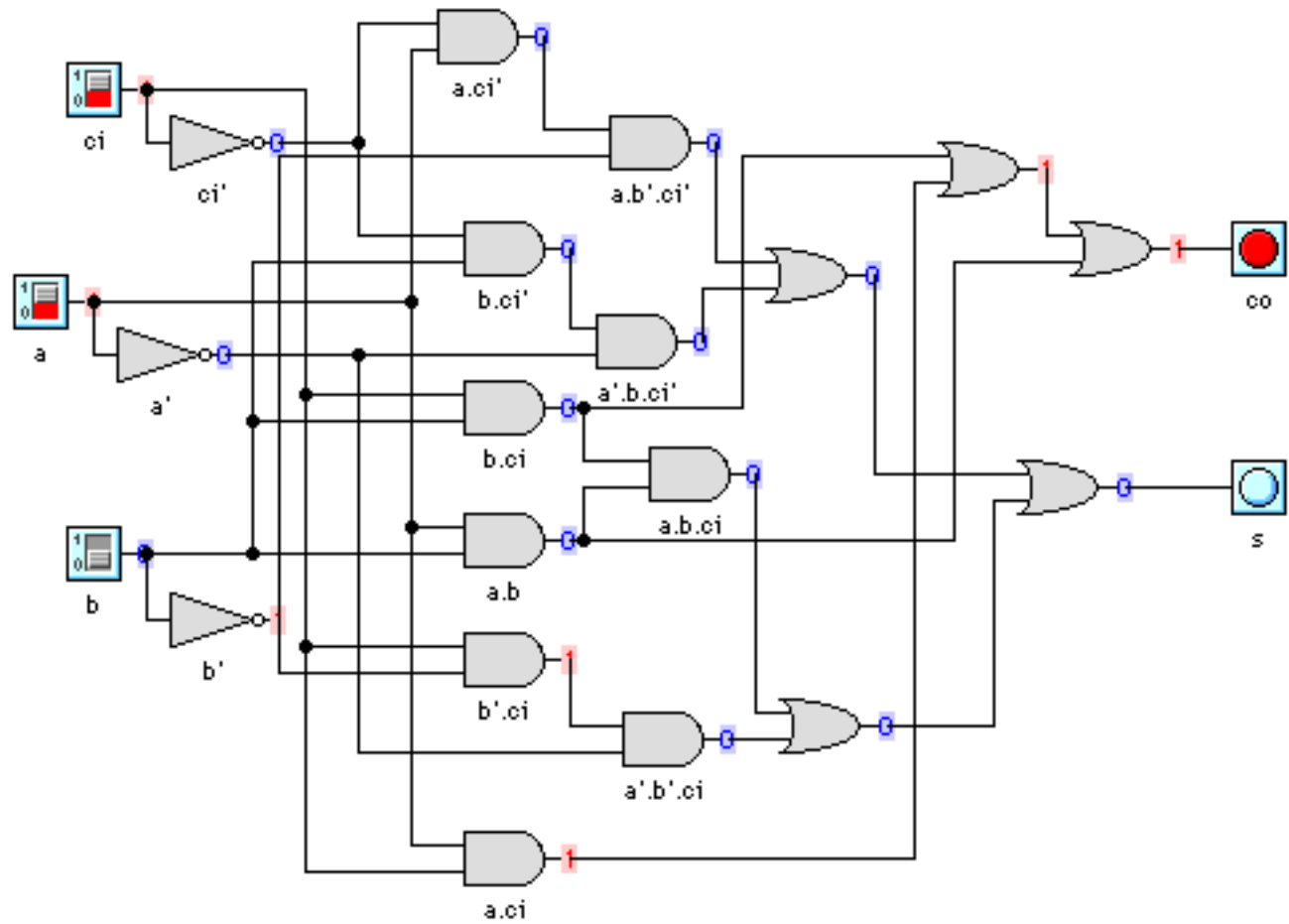
- ▷ 2 outputs:  $s$  (Sum),  $c_o$  (CarryOut — to more significant bit)

- Truth table:

$a$	$b$	$c_i$	$c_o$	$s$	$c_o$ term	$s$ term
0	0	0	0	0		
1	0	0	0	1		$\bar{a}\bar{b}\bar{c}_i$
0	1	0	0	1		$\bar{a}b\bar{c}_i$
1	1	0	1	0	$ab\bar{c}_i$	
0	0	1	0	1		$\bar{a}\bar{b}c_i$
1	0	1	1	0	$a\bar{b}c_i$	
0	1	1	1	0	$\bar{a}bc_i$	
1	1	1	1	1	$abc_i$	$abc_i$

- Logic equations are on a later slide

# 1-BIT FULL ADDER



## REAL ADDERS DON'T USE GATES

- Gate-level design uses too many transistors
  - ▷ Some operations that need to be performed in order to reproduce the truth table for a gate have to be undone by the transistors in the next gate
  - ▷ For a 1-bit full adder, the gate-level design on the previous slide requires 66 transistors
  - ▷ The transistor-level design on the following slide requires only 28 transistors



**LOGIC CIRCUITS (9)**

- Intuitive approach to constructing a one-bit full adder from one-bit half adders:

- ▷ Add the inputs  $a$  and  $b$  using a half adder

- Outputs of this stage:

$$s' = a \cdot \bar{b} + \bar{a} \cdot b$$

$$c' = a \cdot b$$

- ▷ Add the CarryIn bit  $c_i$  to  $s'$  using another half adder

- Outputs of this stage:

$$s = (a \cdot \bar{b} + \bar{a} \cdot b) \cdot \bar{c}_i + \overline{(a \cdot \bar{b} + \bar{a} \cdot b)} \cdot c_i$$

$$c'' = c_i \cdot (a \cdot \bar{b} + \bar{a} \cdot b)$$

- ▷ Set the CarryOut bit  $c_o$  if either  $c'$  or  $c''$  is set:  $c_o = c' + c''$

**LOGIC CIRCUITS (10)**

- Logic equations for a one-bit full adder from the truth table:
  - ▷ 4 input combinations give true outputs for Sum ( $s$ )
  - ▷ SOP logic equation for  $s$ :

$$s = a \cdot \bar{b} \cdot \bar{c}_i + \bar{a} \cdot b \cdot \bar{c}_i + \bar{a} \cdot \bar{b} \cdot c_i + a \cdot b \cdot c_i$$

- ▷ 4 input combinations give true outputs for CarryOut ( $c_o$ )
  - ▷ SOP logic equation for  $c_o$ :

$$c_o = a \cdot b + a \cdot c_i + b \cdot c_i$$

**LOGIC CIRCUITS (11)**

- Logic equations for a one-bit full adder, considered as a combination of two half adders:

▷ Sum ( $s$ ):

$$\begin{aligned} s &= (a \cdot \bar{b} + \bar{a} \cdot b) \cdot \bar{c}_i + \overline{(a \cdot \bar{b} + \bar{a} \cdot b)} \cdot c_i \\ &= (a \cdot \bar{b} + \bar{a} \cdot b) \cdot \bar{c}_i + \overline{(a \cdot \bar{b})} \cdot \overline{(\bar{a} \cdot b)} \cdot c_i \\ &= (a \cdot \bar{b} + \bar{a} \cdot b) \cdot \bar{c}_i + (\bar{a} + b) \cdot (a + \bar{b}) \cdot c_i \\ &= (a \cdot \bar{b} + \bar{a} \cdot b) \cdot \bar{c}_i + (\bar{a} \cdot a + \bar{a} \cdot \bar{b} + a \cdot b + \bar{b} \cdot b) \cdot c_i \\ &= (a \cdot \bar{b} + \bar{a} \cdot b) \cdot \bar{c}_i + (\bar{a} \cdot \bar{b} + a \cdot b) \cdot c_i \end{aligned}$$

▷ Agrees with SOP equation for  $s$  derived from truth table

**LOGIC CIRCUITS (12)**

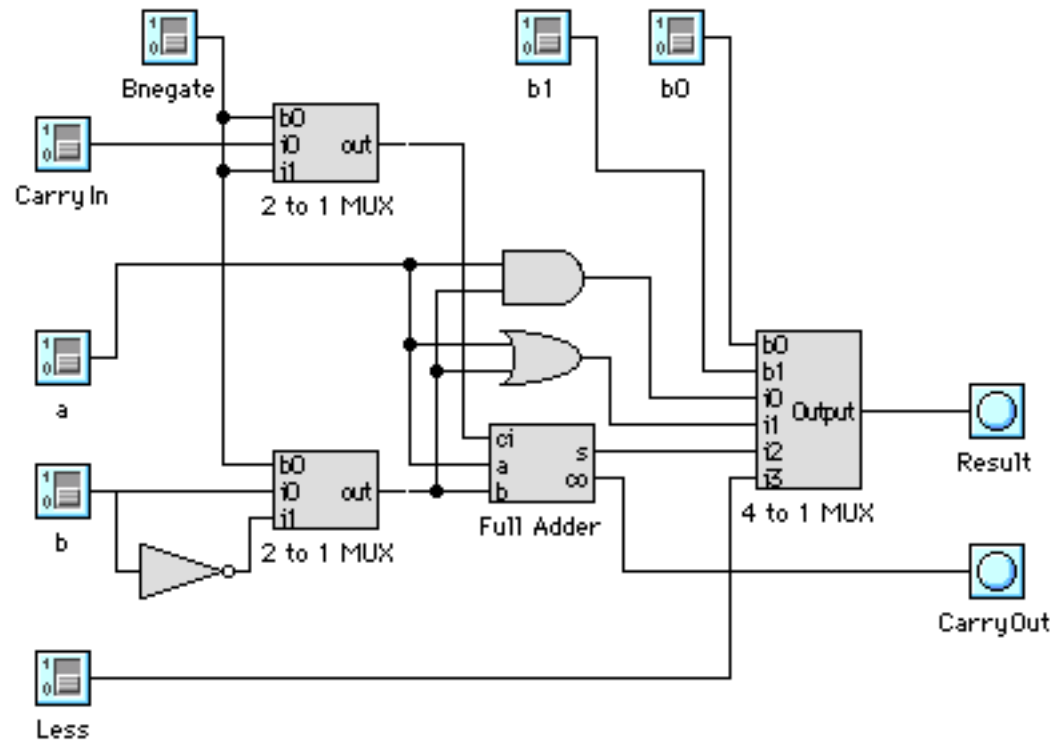
- Logic equations for a one-bit full adder, considered as a combination of two half adders:
  - ▷ For CarryOut ( $c_o$ ), use

$$a \cdot b = a \cdot b + a \cdot b \cdot c_i$$

(the Absorption Theorem) twice:

$$\begin{aligned}c_o &= a \cdot \bar{b} \cdot c_i + \bar{a} \cdot b \cdot c_i + a \cdot b \\&= a \cdot \bar{b} \cdot c_i + \bar{a} \cdot b \cdot c_i + a \cdot b + a \cdot b \cdot c_i \\&= a \cdot (\bar{b} + b) \cdot c_i + \bar{a} \cdot b \cdot c_i + a \cdot b \\&= a \cdot c_i + \bar{a} \cdot b \cdot c_i + a \cdot b \\&= a \cdot c_i + \bar{a} \cdot b \cdot c_i + a \cdot b + a \cdot b \cdot c_i \\&= a \cdot c_i + (\bar{a} + a) \cdot b \cdot c_i + a \cdot b \\&= a \cdot c_i + b \cdot c_i + a \cdot b \quad (\text{agrees with truth-table result})\end{aligned}$$

# 1-BIT ALU



**ALU CONTROL SIGNALS (1)**

- Three signals are necessary for the simple ALU of P & H, Chapter 4
  - ▷ Bnegate: Asserts Binvert and CarryIn
  - ▷ Operation: Selects the output signal
    - 0 for **and**, 1 for **or** or **slt**,  $2_{10}$  for **add** or **sub**

ALU Control Signals			
Bnegate	Operation		MIPS
$b_2$	$b_1$	$b_0$	Instructions
0	0	0	<b>and</b>
0	0	1	<b>or</b>
0	1	0	<b>add</b>
1	1	0	<b>sub</b>
1	1	1	<b>slt</b>

## ALU CONTROL SIGNALS (2)

- ALU control signals for the single-clock-cycle implementation in Patterson & Hennessy, Chapter 5
  - ▷ Derived from the instruction fields Opcode (bits 31–26) and Function (bits 5–0)
  - ▷ ALUOp0
    - ALUOp0 = 1 for R-type instructions (Opcode = 0)
    - ALUOp0 = 0 for all other instructions (Opcode  $\neq$  0)
  - ▷ ALUOp1
    - ALUOp1 = 0 for all R-type instructions (Opcode = 0) and for `lw` (Opcode = 100011), `sw` (Opcode = 101011)
    - ALUOp1 = 1 for `beq` instruction (Opcode = 000100)

## HAZARDS (1)

- A **hazard** is a condition in a *logically correct* digital circuit or computer program that may lead to a logically incorrect output
- Static hazards: Output should stay constant, but doesn't
  - ▷ Static 1 hazard: Output should be a constant 1, but when one input is changed, the output drops to 0 and then recovers to 1
    - Cannot occur in a POS implementation
  - ▷ Static 0 hazard: Output should be a constant 0, but when one input is changed, the output rises to 1 and then drops back to 0
    - Cannot occur in a SOP implementation

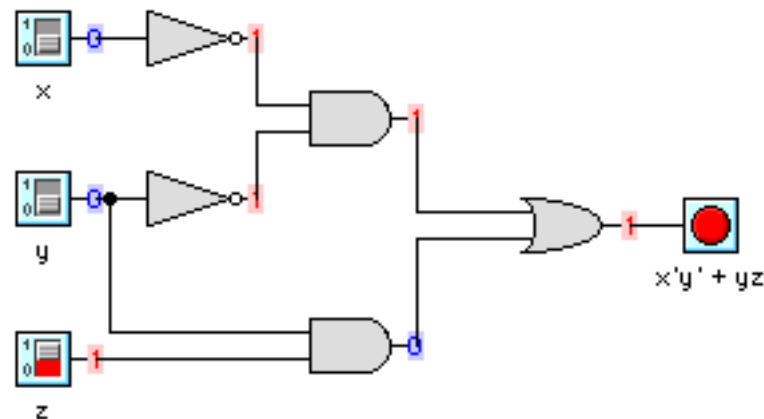
## HAZARDS (2)

- **Why do hazards matter?**

- ▷ The output of a hazard-prone circuit or program depends on conditions other than the inputs and the state
- ▷ The signal passed to another circuit by a hazard-prone circuit depends on exactly when the output is read
- ▷ In edge-triggered logic circuits, a momentary glitch resulting from a hazard can be converted into an erroneous output

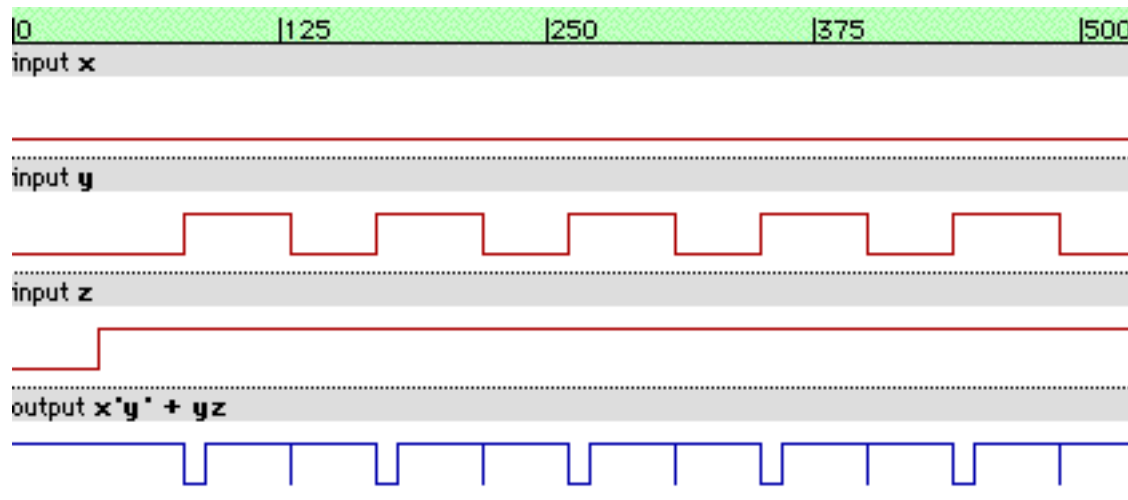
**HAZARDS (3)**

- The circuit below for  $\bar{x}\bar{y} + yz$  has a static 1 hazard
  - ▷ If the input  $y$  is changed from 0 to 1, control of the output of the OR gate shifts from one AND gate to the other
  - ▷ Any difference in delays between the two AND gates will result in a glitch in the output of the OR



## HAZARDS (4)

- The timing diagram below shows the inputs and outputs of a circuit for  $\bar{x}\bar{y} + yz$  with a static 1 hazard



## HAZARDS (5)

- Static 1 hazard detection using a Karnaugh map:
  - ▷ Reduce the logic function to a minimal sum of prime implicants
  - ▷ A Karnaugh map that contains adjacent, disjoint prime implicants is subject to a static 1 hazard
    - Adjacent prime implicants: Only one variable needs to change value to move from one prime implicant to the other
    - Disjoint prime implicants
      - ◇ No prime implicant covers cells of both of the disjoint prime implicants
      - ◇ Correspond to AND gates that must both change their outputs when a particular input is changed

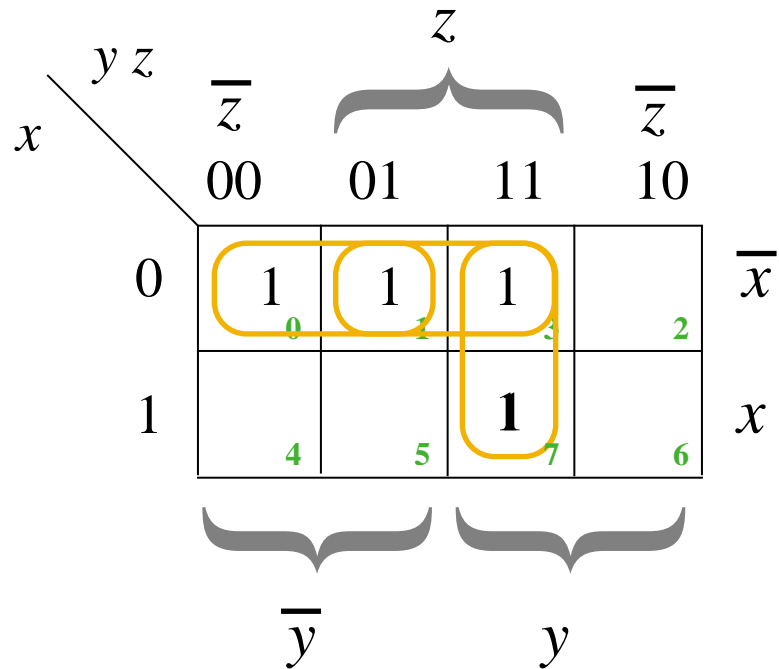
# Hazard detection

$$F(x,y,z) = \Sigma m(0,1,3,7)$$

		$z$				
		$\bar{z}$	}		$\bar{z}$	
$x$	$y z$	$\bar{z}$	01	11	$\bar{z}$	
		00			10	
0		1	1	1		$\bar{x}$
1				1		$x$
		4	5	7	6	
		}		}		
		$\bar{y}$		$y$		

$$F(x,y,z) = \bar{x}\bar{y} + yz$$

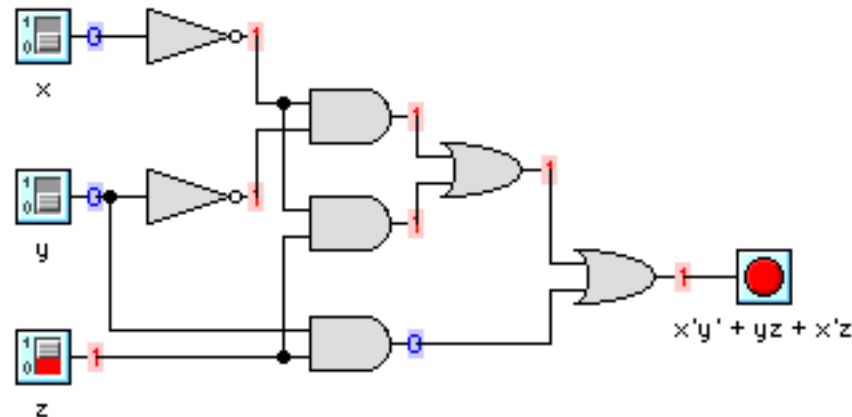
# Hazard elimination



$$F(x, y, z) = \bar{x}\bar{y} + yz + \bar{x}z$$

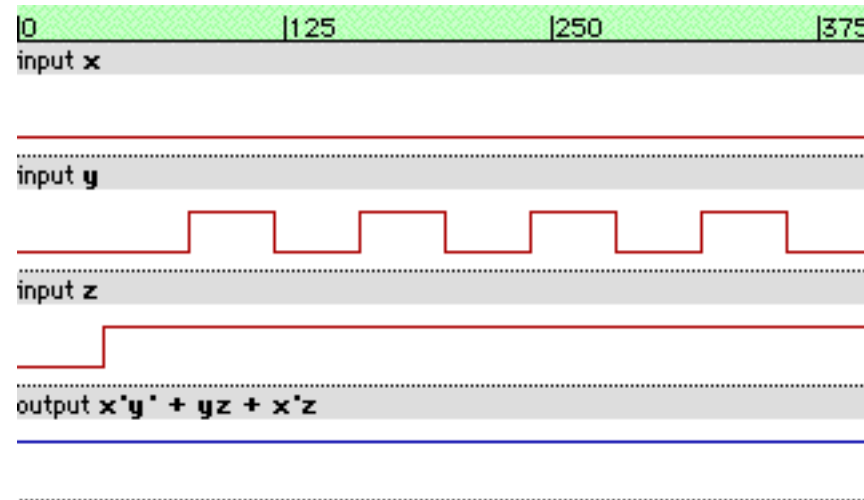
## HAZARDS (6)

- The circuit below for  $\bar{x}\bar{y} + yz$  has no static 1 hazard



## HAZARDS (7)

- The timing diagram below shows the inputs and outputs of a revised circuit for  $\bar{x}\bar{y} + yz$  with no static 1 hazard



## SEQUENTIAL LOGIC CIRCUITS (1)

- The outputs  $z_1, \dots, z_m$  of a **sequential** logic circuit depend on:
  - ▷ The **inputs**  $x_1, \dots, x_n$
  - ▷ Internal logical variables  $y_1, \dots, y_r$  (the **present state**)
  - ▷ The **next state**  $y_1^*, \dots, y_r^*$  depends on the inputs and the present state:

$$y_j^* = h_j(x_1, \dots, x_n, y_1, \dots, y_r) \quad [j \in (1 : r)]$$

- Contrast with a combinational logic circuit, where the outputs depend only on the inputs:

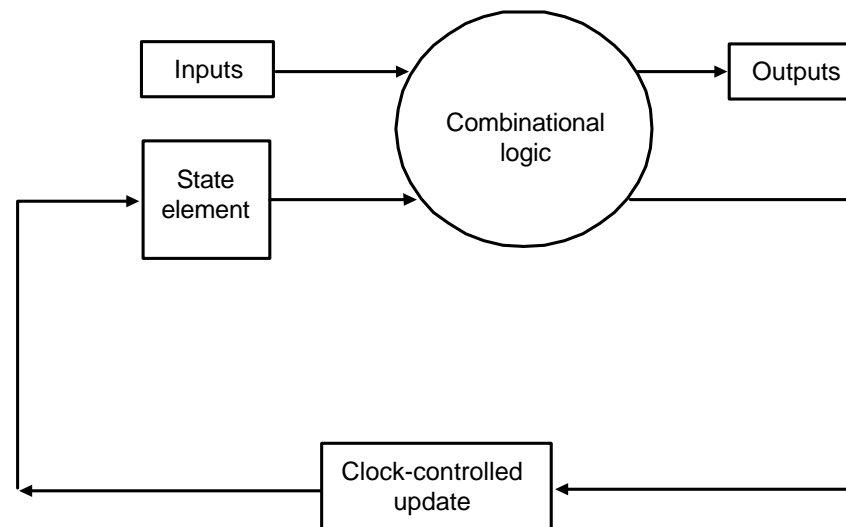
$$z_i = f_i(x_1, \dots, x_n) \quad [i \in (1 : m)]$$

- For a sequential logic circuit, the outputs depend on the inputs *and* the present state:

$$z_i = g_i(x_1, \dots, x_n, y_1, \dots, y_r) \quad [i \in (1 : m)]$$

## SEQUENTIAL LOGIC CIRCUITS (2)

- Huffman model of a sequential circuit:
  - ▷ State element
    - Memory that holds the present state
    - Normally updated at intervals controlled by a clock signal
  - ▷ Combinational logic that implements the Boolean functions  $z_i$  (outputs) and  $y_j^*$  (next state)

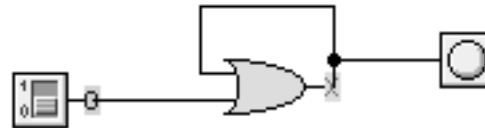


## LATCHES

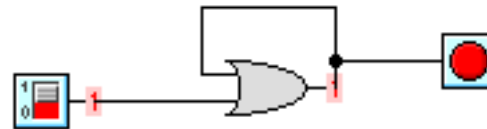
- A **latch** is a sequential logic circuit that is controlled entirely by its **excitation inputs**, *without* a clock signal
  - ▷ A latch is a **bistable device**
    - Can remain indefinitely in either of two stable states
    - For all but the simplest latches, the state can be changed by asserting one of the latch's inputs
    - Used as a 1-bit memory element
  - ▷ Simplest latches
    - Set latch (can be set to 1, but not reset to 0)
    - Reset latch (can be set to 0, but not reset to 1)
  - ▷ Practical latches
    - Set-reset latch
    - Delay latch

**SET LATCH**

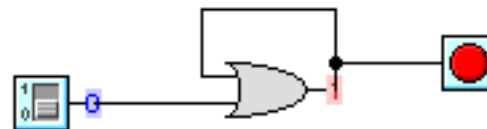
- A **set latch** can be set to 1, but not reset to 0
  - ▷ Initial state: Input = 0, output =  $\times$  (undefined)



- ▷ Input asserted  $\Rightarrow$  output = 1:

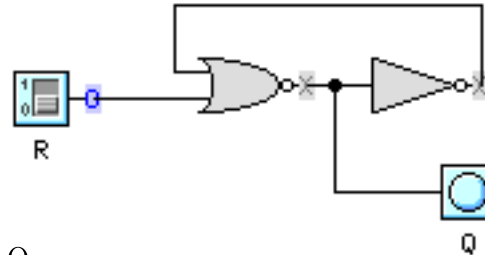


- ▷ Output is now independent of input:

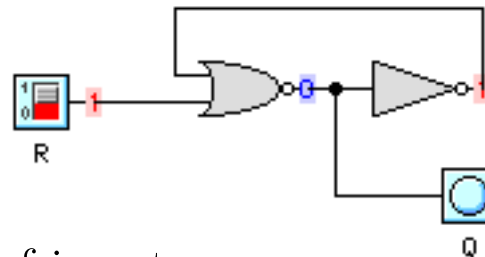


**RESET LATCH**

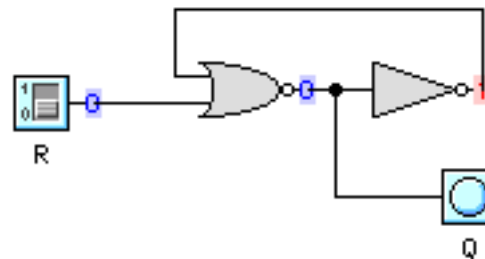
- A **reset latch** can be reset to 0, but not set to 1
  - ▷ Initial state: Input = 0, output =  $\times$  (undefined)



- ▷ Input asserted  $\Rightarrow$  output = 0:

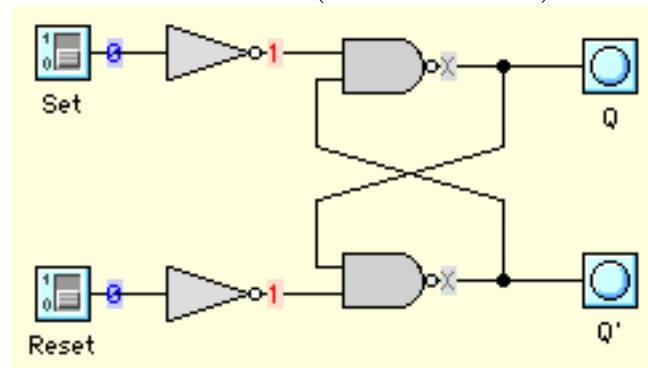


- ▷ Output is now independent of input:

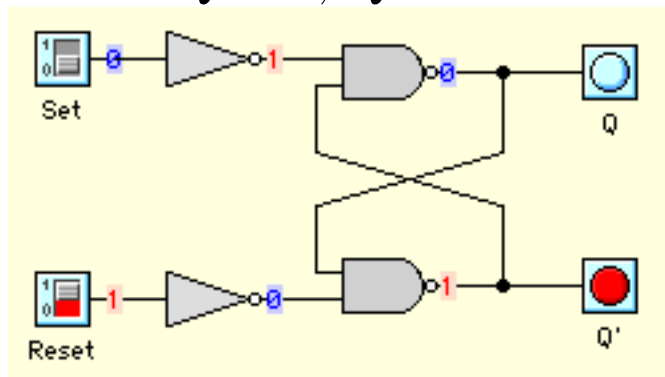


## SET-RESET (SR) LATCH (1)

- Initial state: Input = 0, outputs =  $\times$  (undefined)

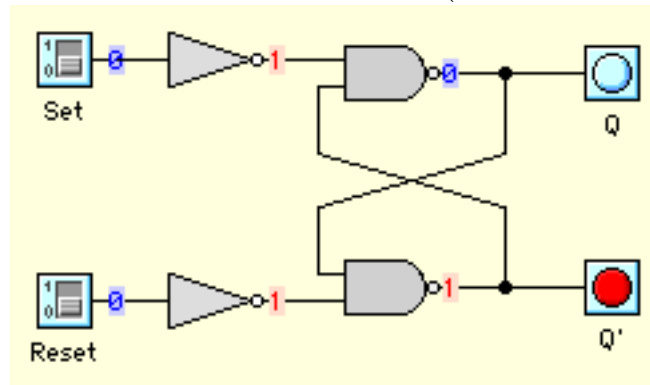


- $R = 1, S = 0 \Rightarrow$  state in which  $Q = 0, \bar{Q} = 1$ :

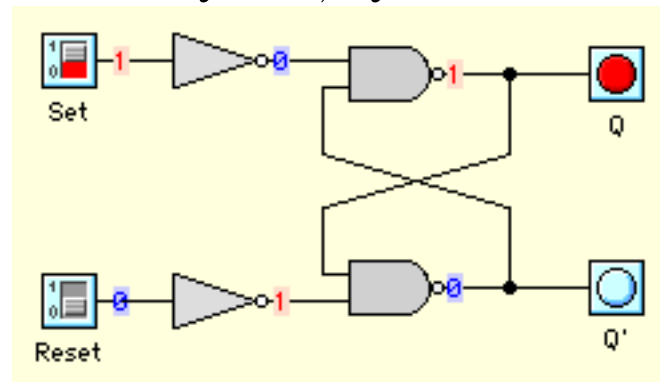


## SET-RESET (SR) LATCH (2)

- Next state: Inputs = 0, outputs unchanged (storage of previous state)

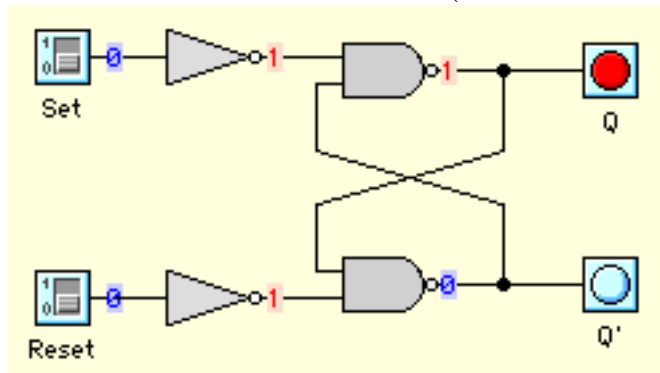


- $R = 0, S = 1 \Rightarrow$  state in which  $Q = 1, \bar{Q} = 0$ :

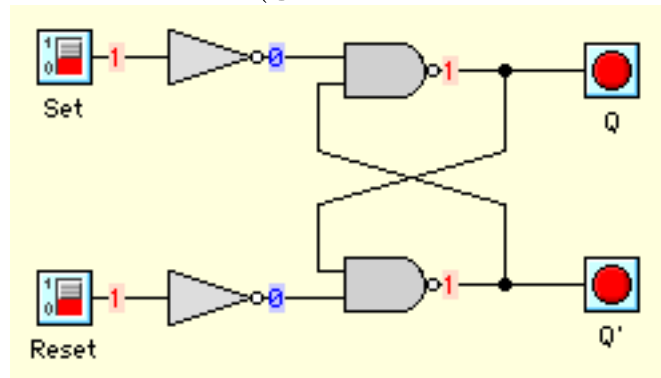


## SET-RESET (SR) LATCH (3)

- Next state: Inputs = 0, outputs unchanged (storage of previous state)

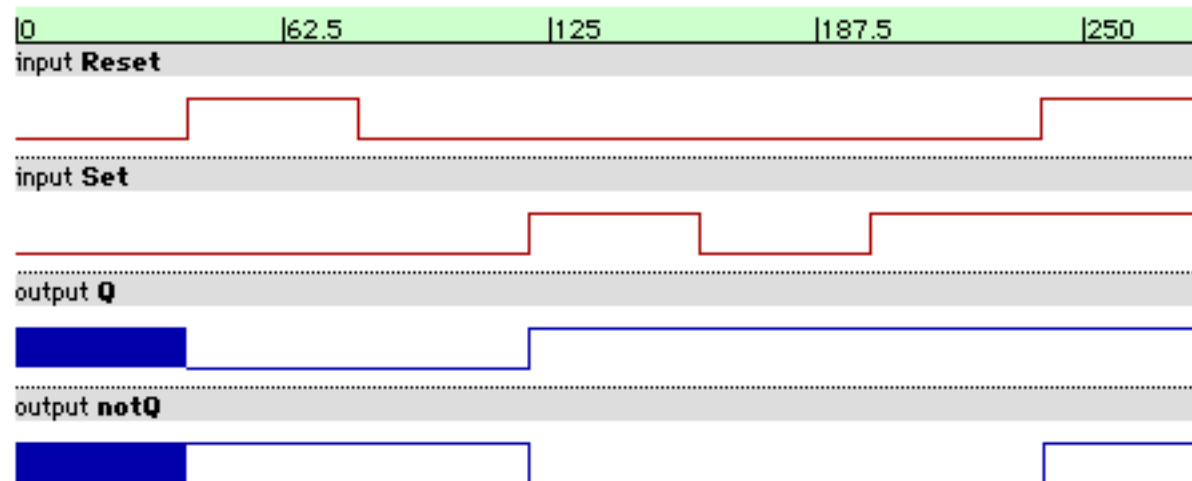


- $R = 1, S = 1 \Rightarrow$  **race condition** (gate with shortest delay “wins”):



## SET-RESET (SR) LATCH (4)

- Timing diagram:

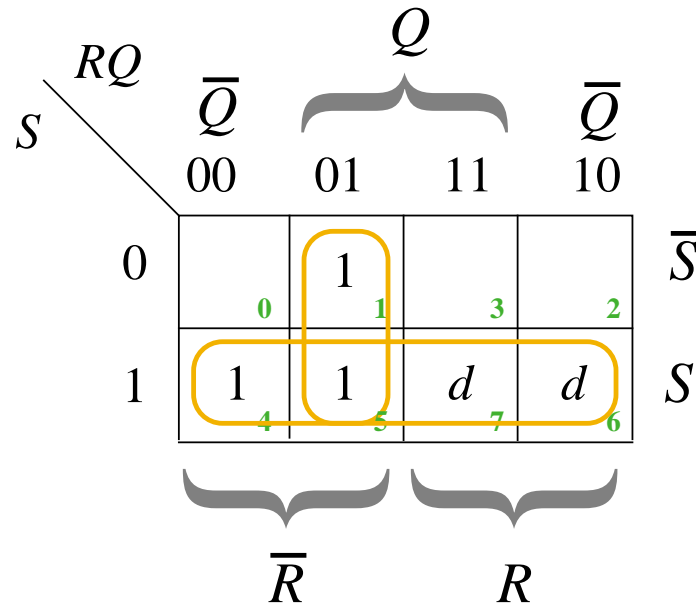


**S-R LATCH EXCITATION TABLE**

S-R Latch Excitation Table						
Inputs		Old State		Next State		Comments
<i>S</i>	<i>R</i>	<i>Q</i>	$\overline{Q}$	<i>Q</i> *	$\overline{Q}$ *	
0	0	0	1	0	1	No change (storage state)
0	0	1	0	1	0	
0	1	0	1	0	1	Reset
0	1	1	0	0	1	
1	0	0	1	1	0	Set
1	0	1	0	1	0	
1	1	0	1	×	×	Race condition when $S, R \rightarrow 1$
1	1	1	0	×	×	

# KARNAUGH MAP OF S-R LATCH EXCITATION TABLE

- Karnaugh map showing the next state,  $Q^*$ , as a function of the inputs  $R$ ,  $S$  and the present state,  $Q$ :



- ▷ The “don’t cares” ( $d$ ) can be covered by any minterm, because the outputs  $Q$  and  $\bar{Q}$  are undefined
- ▷ Choose the minterms that give the simplest logic circuit

**EQUATION FOR NEXT STATE OF S-R LATCH**

- The state of an S-R latch is specified uniquely by  $Q$ 
  - ▷ Characteristic equation:

$$Q^* = S + \bar{R} \cdot Q$$

- Gives next state ( $Q^*$ ) in terms of present state ( $Q$ ) and excitation inputs ( $S, R$ )
- Can be derived from excitation table
- ▷ When  $S = R = 1$ , both  $Q$  and  $\bar{Q}$  are set to logical 1 until at least one input changes
  - The next state ( $Q^*$ ) is undefined
  - The characteristic equation does not apply to this case

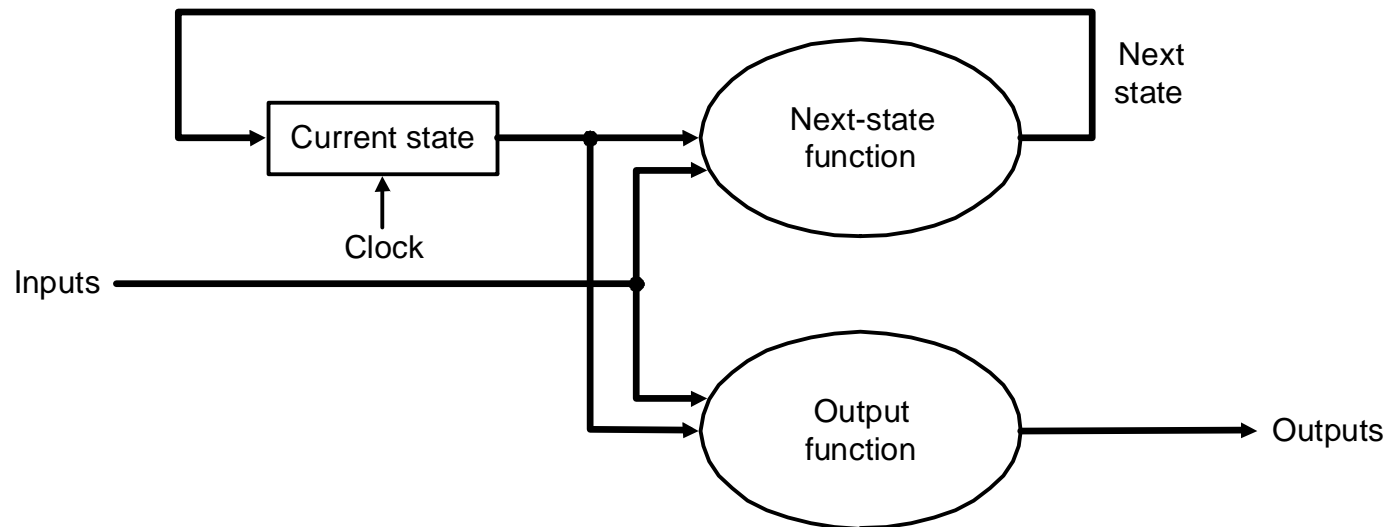
**S-R LATCH STATE TABLE**

S-R Latch State Table							
Inputs		Old State		Next State		Outputs	
$S$	$R$	$Q$	$\bar{Q}$	$Q^*$	$\bar{Q}^*$	$Q^*$	$\bar{Q}^*$
0	0	0	1	0	1	0	1
0	0	1	0	1	0	1	0
0	1	0	1	0	1	0	1
0	1	1	0	0	1	0	1
1	0	0	1	1	0	1	0
1	0	1	0	1	0	1	0
1	1	0	1	×	×	×	×
1	1	1	0	×	×	×	×

- For the S-R latch, the outputs are the same as the next state variables
  - ▷ This isn't true for the state table of every sequential circuit

## FINITE STATE MACHINES (1)

- A **finite state machine** is a conceptual tool used to describe the computational functioning of a sequential logic circuit without specifying the implementation

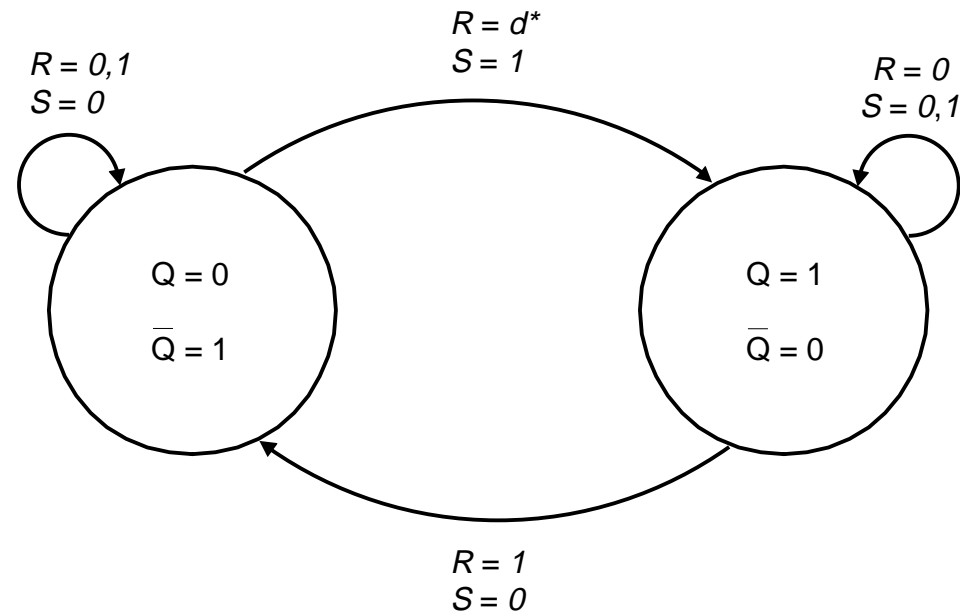


## FINITE STATE MACHINES (2)

- A **state diagram** is a certain kind of directed graph
  - ▷ The nodes (vertexes) represent states of the machine
    - A state is defined by the values of the internal logical variables
    - Each node in a state diagram is labeled with the values that define the state that corresponds to the node
  - ▷ The edges represent the state transitions
    - Each edge in a state diagram is labeled with the inputs that cause the state transition that corresponds to the edge

## FINITE STATE MACHINES (3)

- The state diagram for the S-R latch:

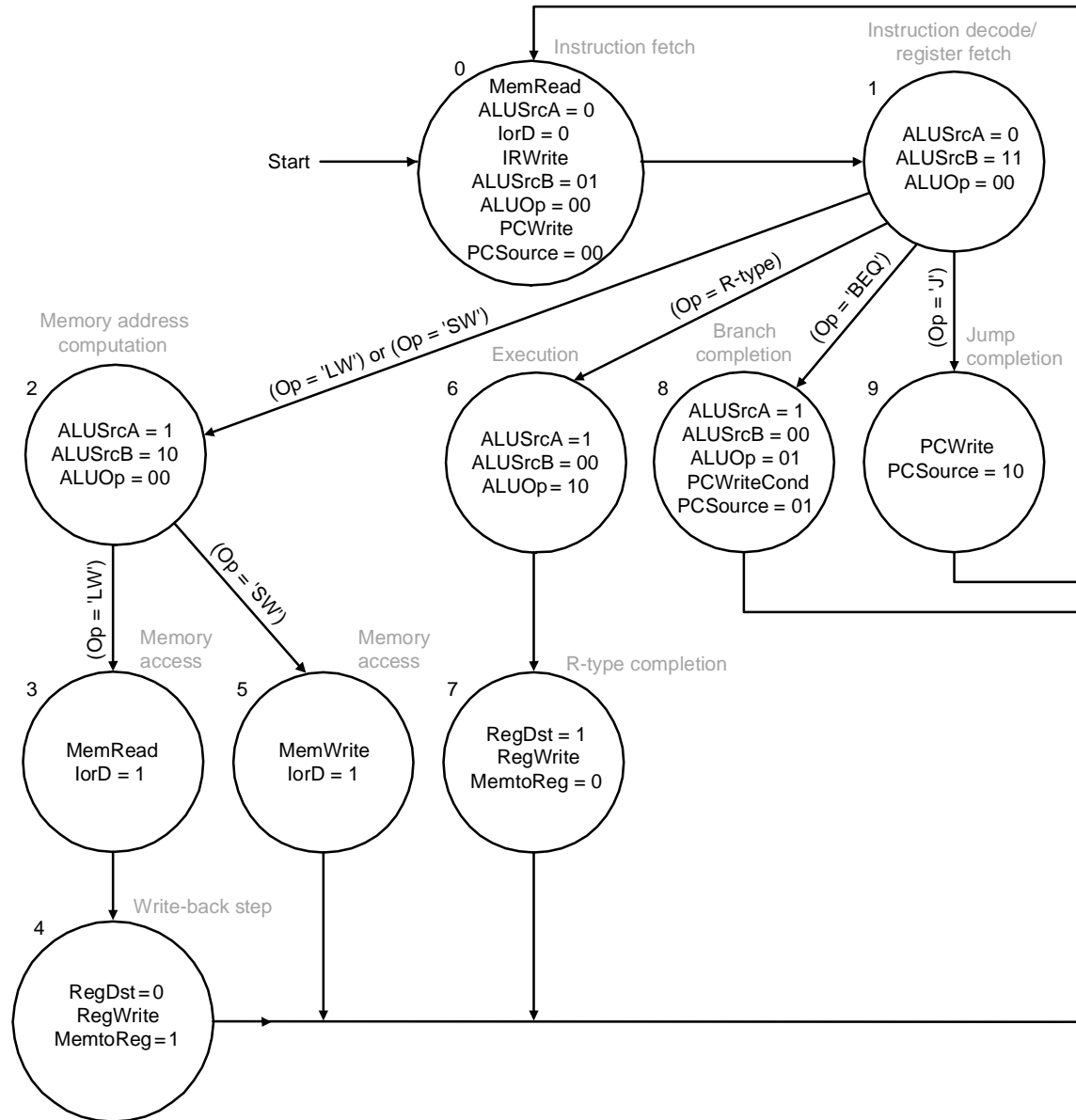


\*  $R = 1$  when  $S = 1$  gives a race condition

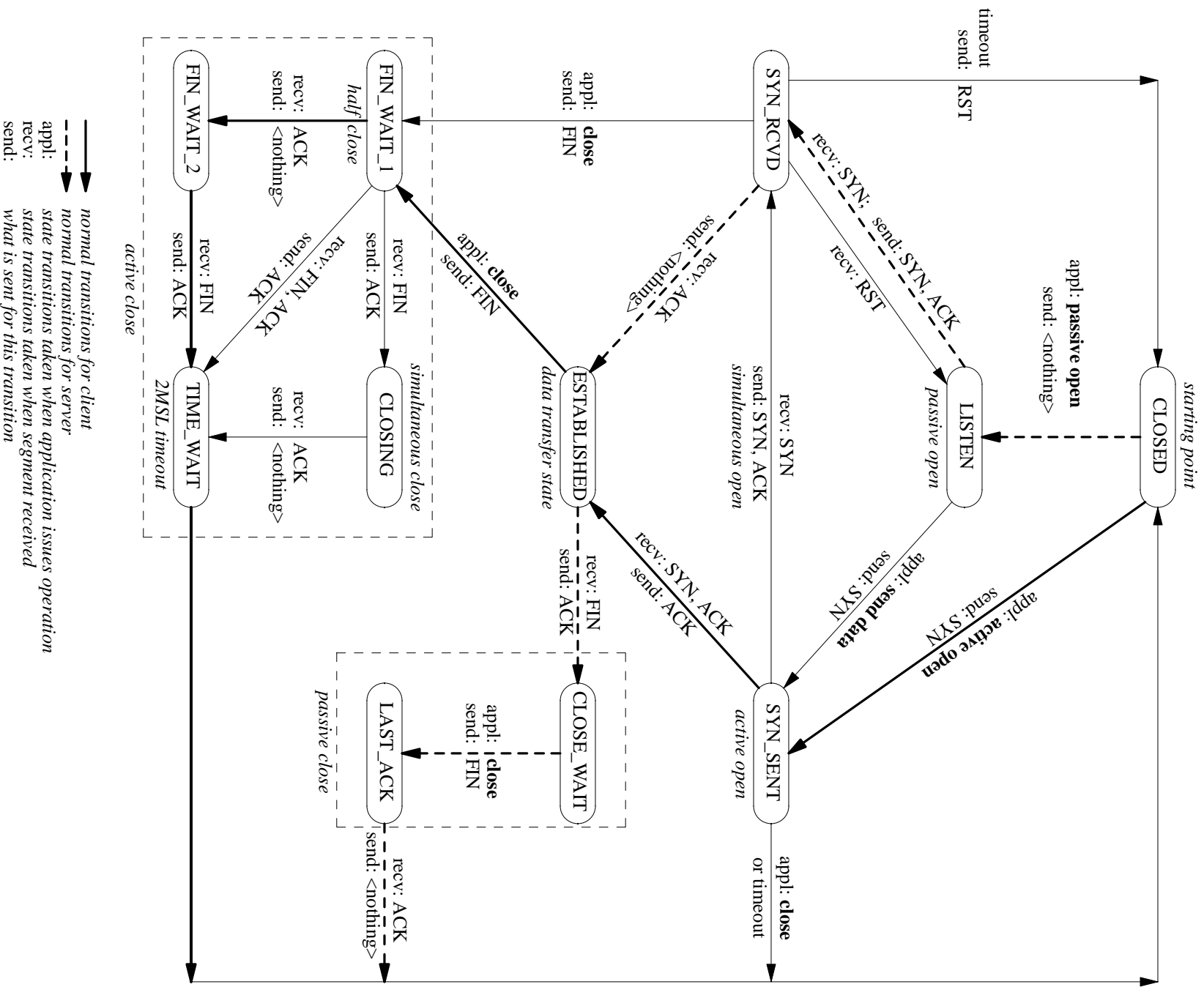
## FINITE STATE MACHINES (4)

- Finite state machines (FSMs) are used in industry to define the functions of a sequential logic circuit or a computer program without specifying the details of the implementation
- A FSM that describes hardware:
  - ▷ The control functions for the simple ALU in P&H, Chapter 5
- FSMs that are usually implemented in software:
  - ▷ The Transmission Control Protocol (TCP)
    - You invoke this protocol whenever you click on a link in a World Wide Web page
  - ▷ The Point-to-Point Protocol (PPP)
    - You invoke this protocol whenever you start Dial-Up Networking in Windows

# Finite State Machine for Simple ALU

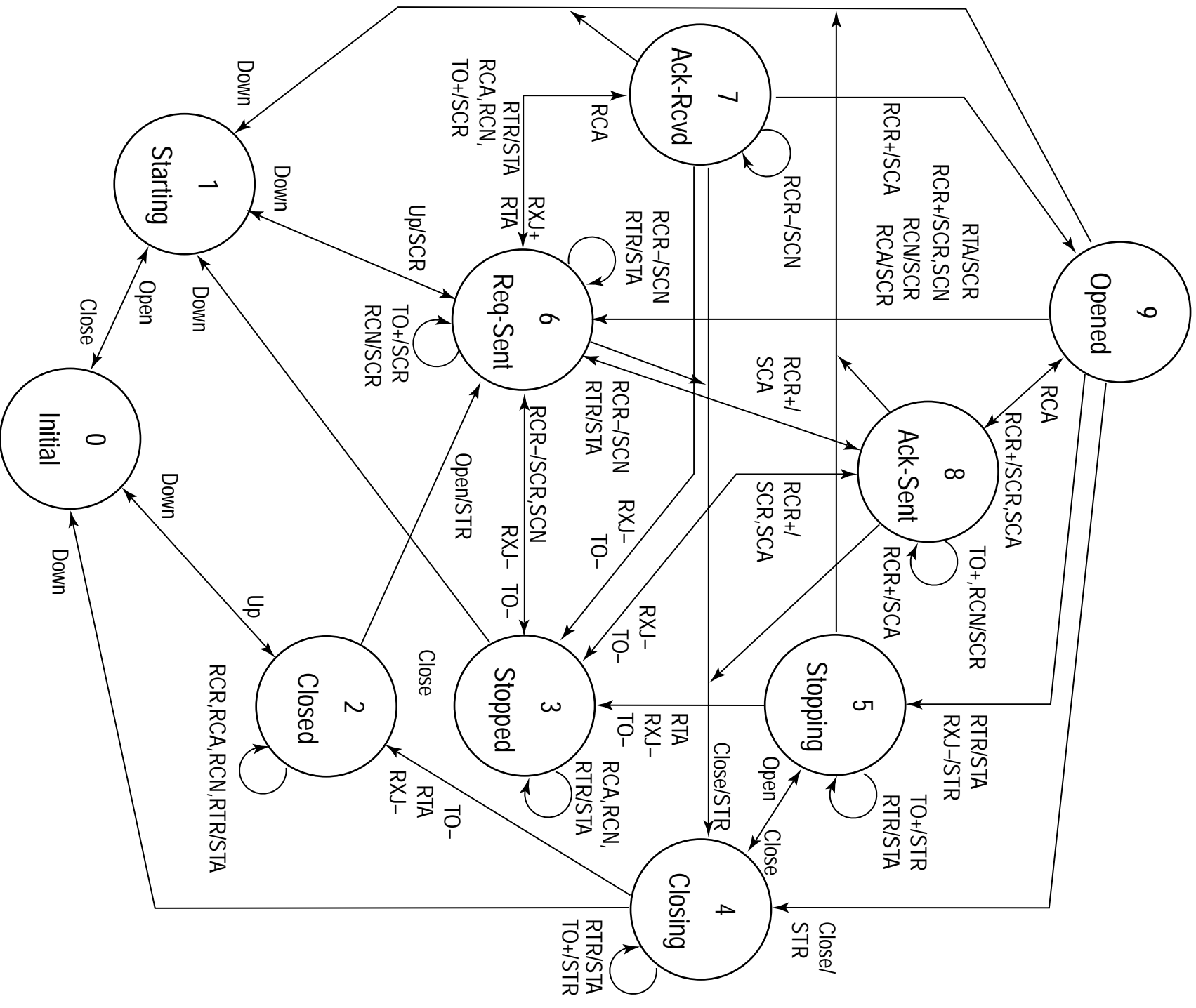


How many bits are needed to number each state uniquely?



## TCP state transition diagram

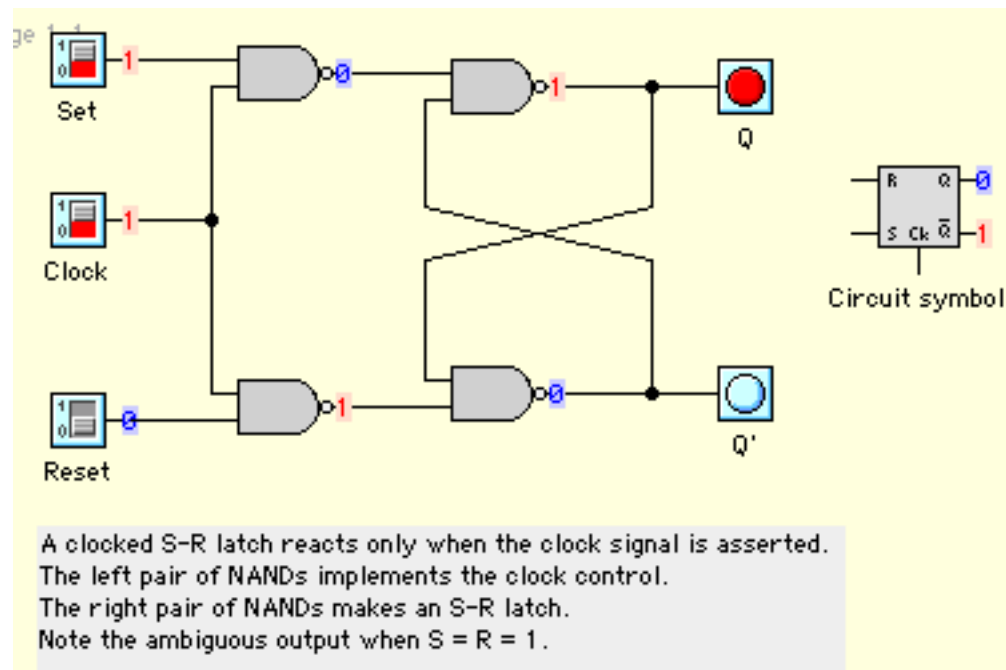
# PPP State Machine



After James Carlson, *PPP: Design and Debugging*

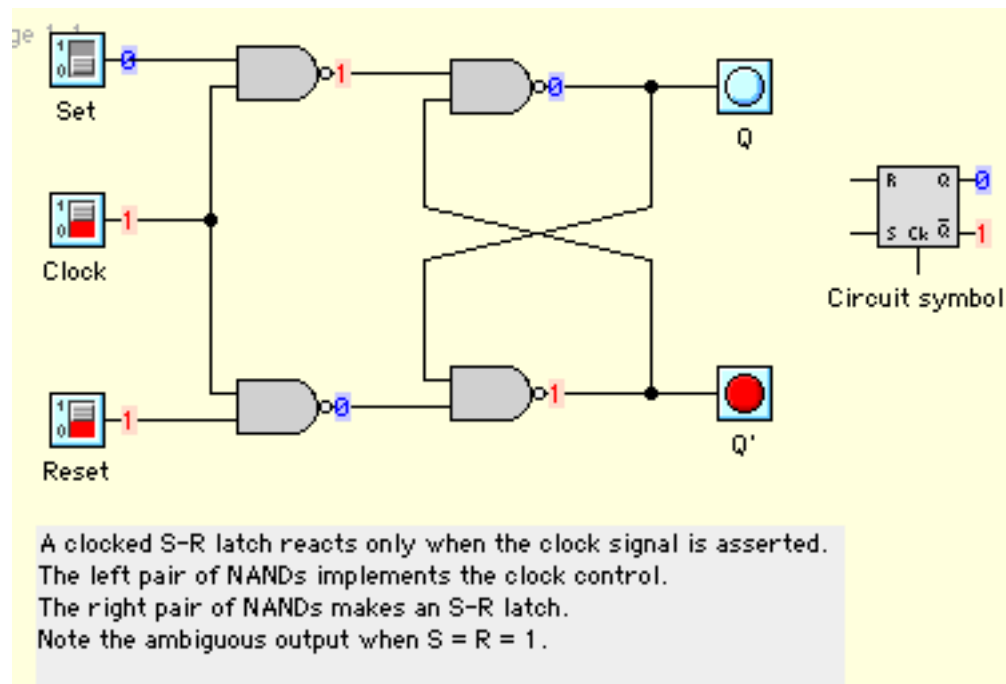
## CSR LATCH (1)

- A **gated SR latch (CSR latch)** is controlled both by its excitation inputs ( $S$ ,  $R$ ) and an “enable” signal ( $C$ )
  - ▷ Notice how the NANDs isolate the S-R latch from the inputs when the clock signal is low, and pass the inputs through to the latch when the clock signal is high



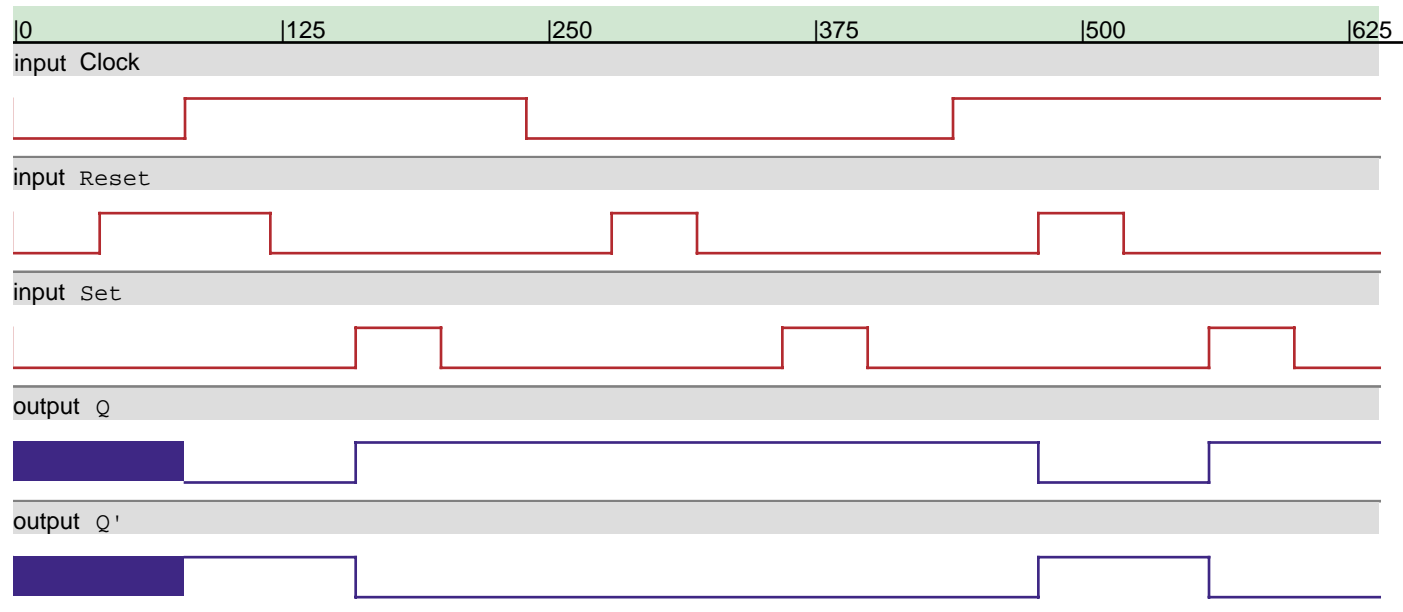
## CSR LATCH (2)

- A **gated SR latch (CSR latch)** is controlled both by its excitation inputs ( $S$ ,  $R$ ) and an “enable” signal ( $C$ )



## CSR LATCH (3)

- Timing diagram:



## CSR LATCH (4)

CSR Latch Excitation Table							
Inputs			Old State		Next State		Comments
$C$	$S$	$R$	$Q$	$\overline{Q}$	$Q^*$	$\overline{Q}^*$	
0	×	×	0	1	0	1	Hold
0	×	×	1	0	1	0	
1	0	0	0	1	0	1	No change (storage state)
1	0	0	1	0	1	0	
1	0	1	0	1	0	1	Reset
1	0	1	1	0	0	1	
1	1	0	0	1	1	0	Set
1	1	0	1	0	1	0	
1	1	1	0	1	×	×	Race condition when $S, R \rightarrow 1$
1	1	1	1	0	×	×	

**CSR LATCH (5)**

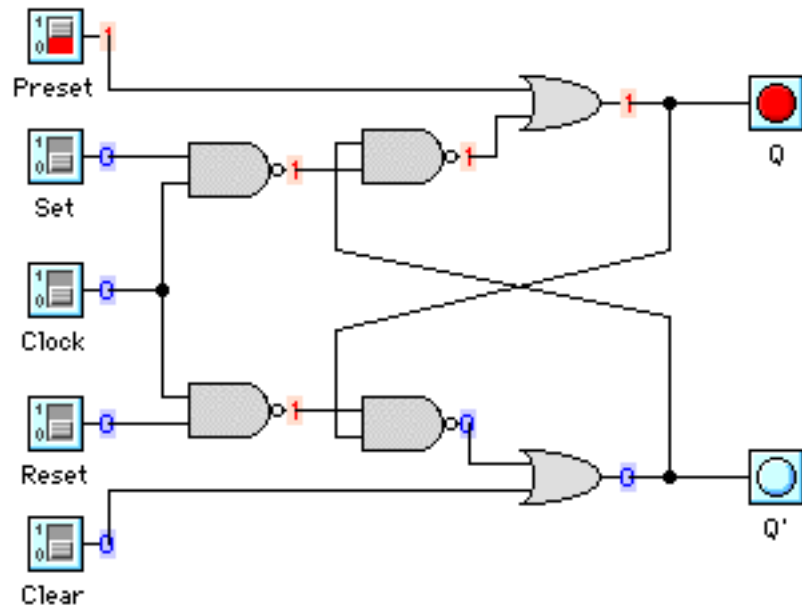
- The state of a CSR latch is specified uniquely by  $Q$

▷ Characteristic equation:

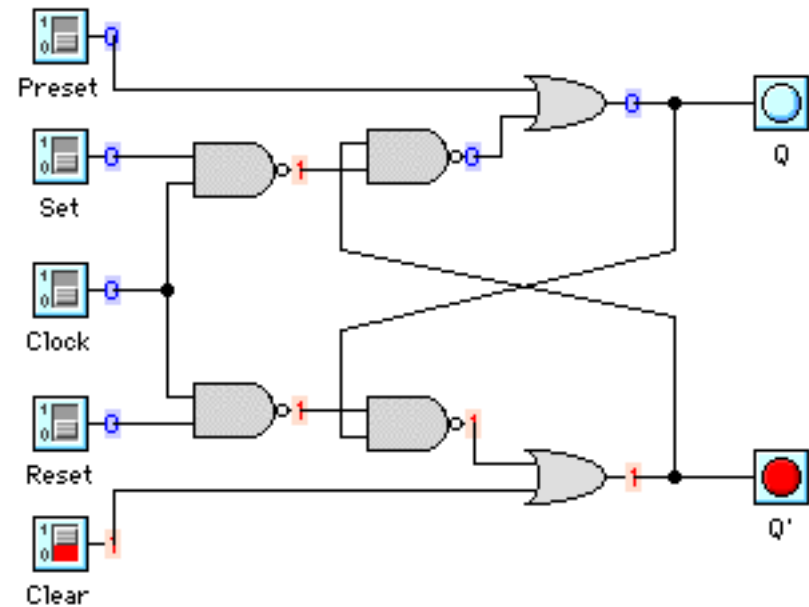
$$Q^* = \overline{C}Q + CS + C\overline{R}Q$$

- Gives next state ( $Q^*$ ) in terms of present state ( $Q$ ), excitation inputs ( $S$ ,  $R$ ), and clock signal ( $C$ )
  - When clock is high ( $C = 1$ ), this is the same as the characteristic equation of the S-R latch ( $Q^* = S + \overline{R}Q$ )
  - When clock is low ( $C = 0$ ), the next state is the same as the present state ( $Q^* = Q$ )
  - Can be derived from excitation table
- ▷ The characteristic equation does not apply when  $S = R = 1$

# CSR LATCH WITH "PRESET" AND "CLEAR" INPUTS



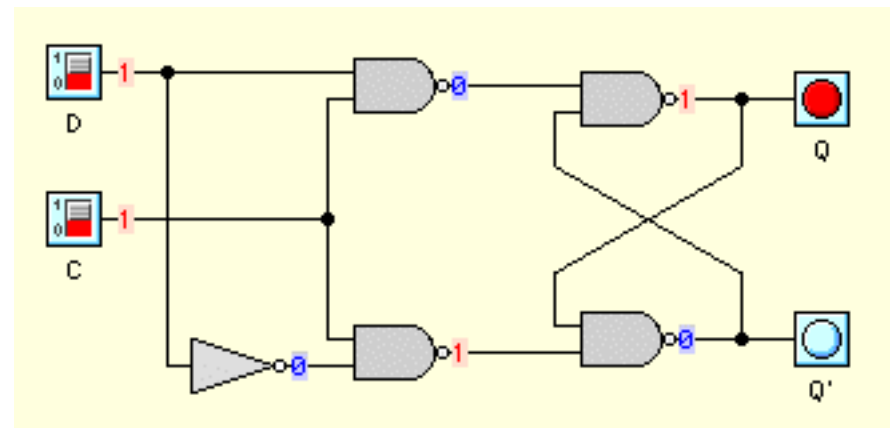
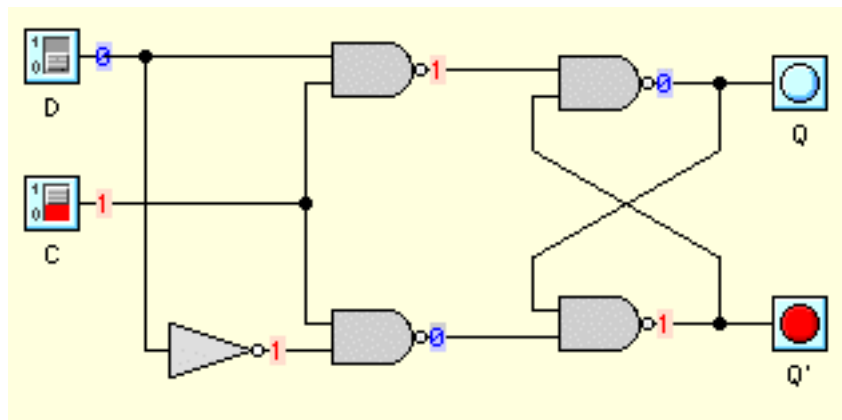
Unlocked Preset and Clear inputs make it possible to set and reset the latch's state asynchronously (independently of the clock signal). This is important at startup time, in order to resolve "unknowns". However,  $S = R = 1$  still produces an ambiguous output ( $Q = Q' = 1$ ).



Unlocked Preset and Clear inputs make it possible to set and reset the latch's state asynchronously (independently of the clock signal). This is important at startup time, in order to resolve "unknowns". However,  $S = R = 1$  still produces an ambiguous output ( $Q = Q' = 1$ ).

# D LATCH (1)

- A **delay latch (D latch)** is a CSR latch in which the inputs are hardwired to be the logical complements of one another



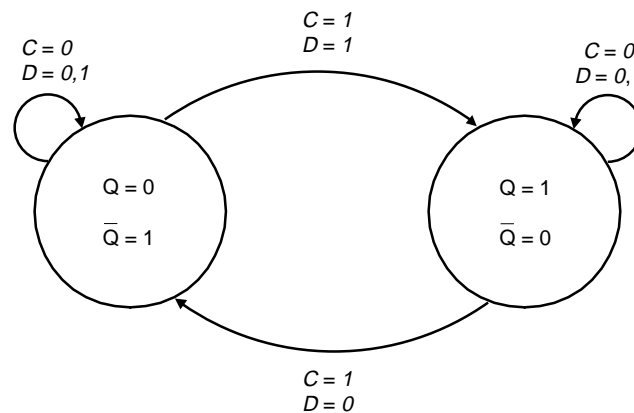
**D LATCH (2)**

- The state of a D latch is specified uniquely by  $Q$

▷ Characteristic equation:

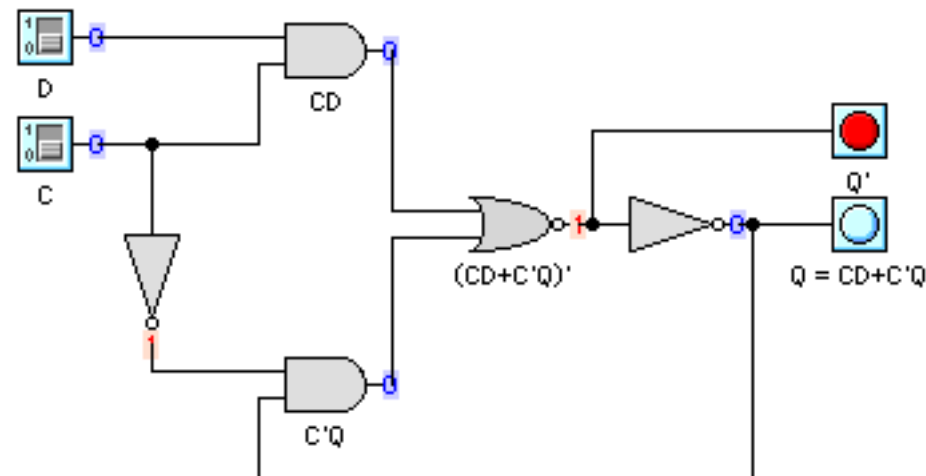
$$Q^* = \overline{C}Q + CD + CDQ = \overline{C}Q + CD$$

- Derived from the characteristic equation of the CSR latch ( $Q^* = \overline{C}Q + CS + C\overline{R}Q$ ) by setting  $S = D$ ,  $R = \overline{D}$
- Can also be derived from excitation table

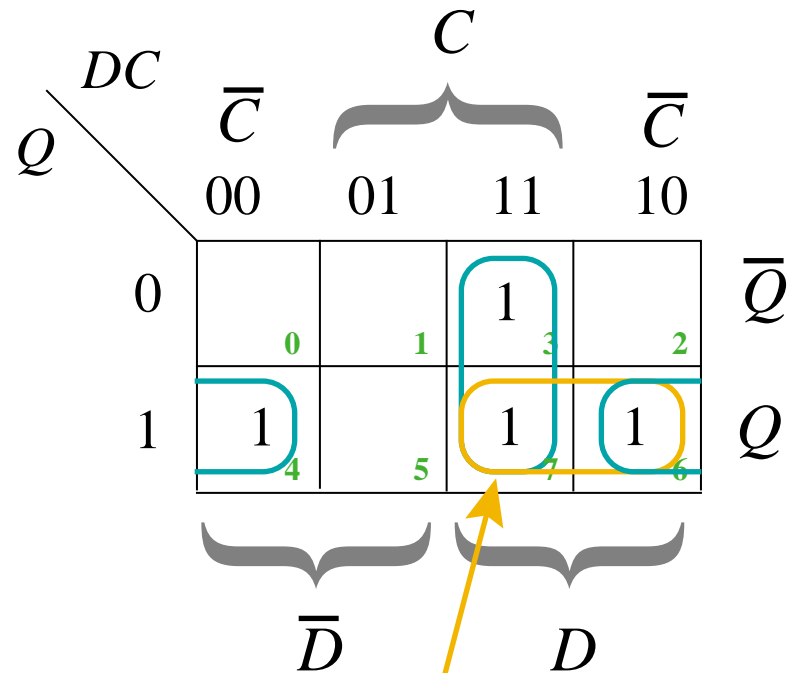


**D LATCH (3)**

- Another realization of a D latch uses a direct implementation of the next-state function  $Q^* = \overline{C}Q + CD$ 
  - ▷ No race condition exists, but there is a static 1 hazard, which can be removed by adding an AND gate that implements the minterm  $DQ$



# Karnaugh map for next state function of a D latch



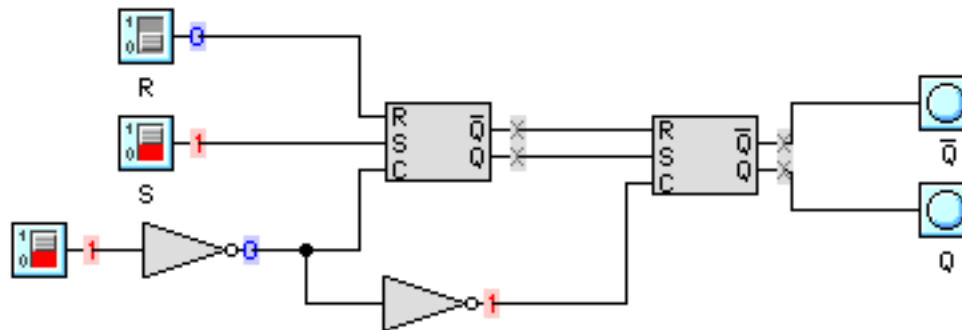
Removes static 1 hazard

## FLIP-FLOPS (1)

- A **flip-flop** is a bistable sequential logic circuit that is controlled both by its excitation inputs *and* a clock signal
  - ▷ The state can be changed by asserting one of the flip-flops's inputs either while the clock signal is asserted, or when the clock signal changes
  - ▷ In useful designs, the output does not change until the leading edge of the next clock signal
- Master-slave flip-flops
  - ▷ Master-slave SR flip-flop
  - ▷ Master-slave D flip-flop

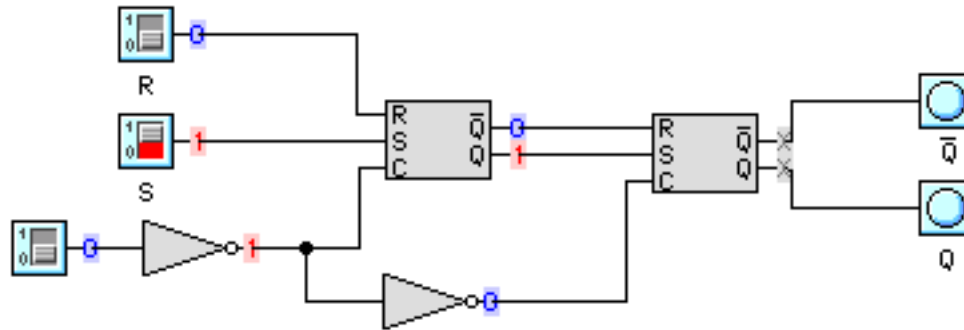
**MASTER-SLAVE S-R FLIP-FLOP (1)**

- For as long as the clock is low, the inputs  $R$  and  $S$ , and the present state  $Q_1$ , control the next state of the first S-R latch
- When the clock goes high, the outputs ( $Q$  and  $\bar{Q}$ ) of the first S-R latch, which are the inputs of the second latch, can change the second latch's state

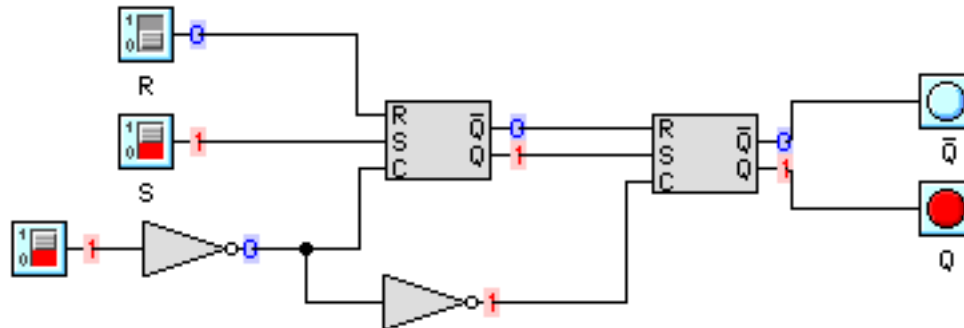


Initial state (just turned on, clock high)

# MASTER-SLAVE S-R FLIP-FLOP (2)

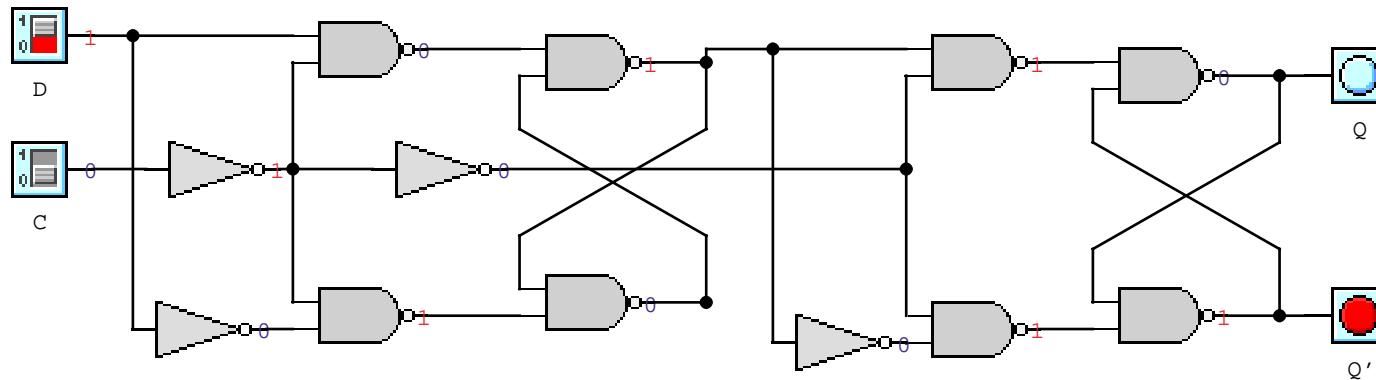


Just after clock goes low



Just after clock returns to high

# MASTER-SLAVE D FLIP-FLOP (1)

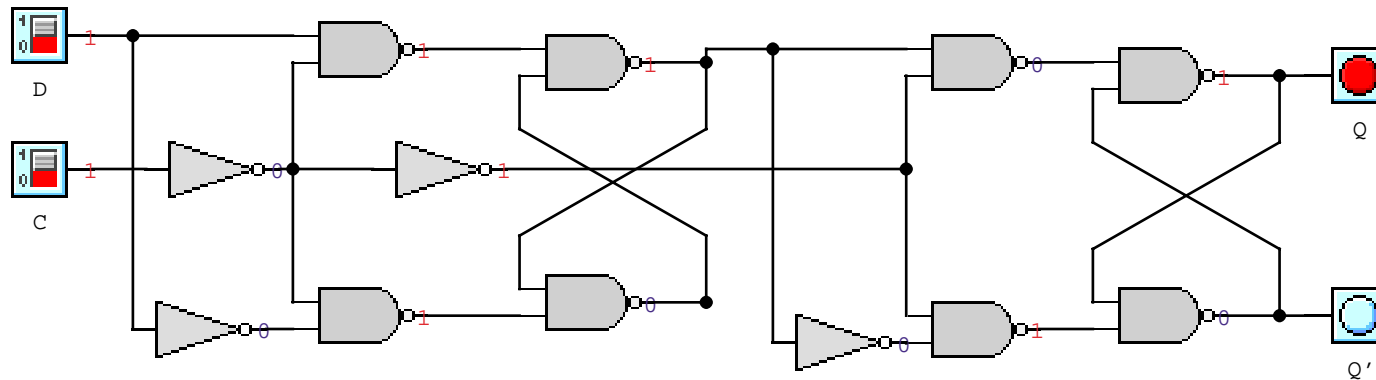


**MASTER LATCH**

**SLAVE LATCH**

Just after  $D$  is asserted while clock is low; present state is  $Q = 0$

# MASTER-SLAVE D FLIP-FLOP (2)

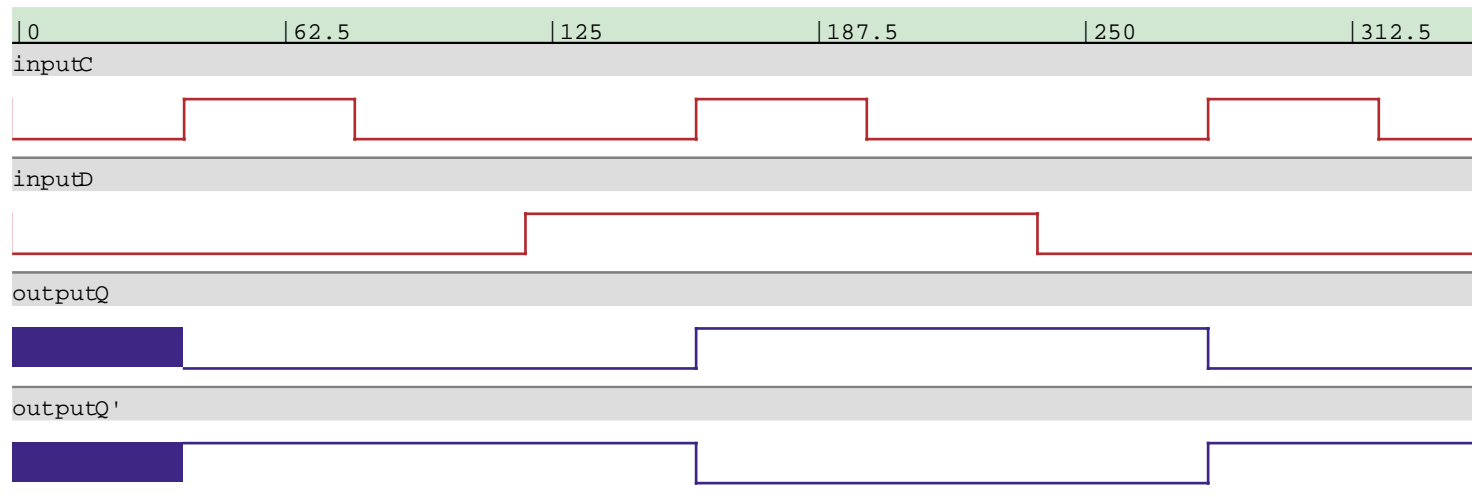


**MASTER LATCH**

**SLAVE LATCH**

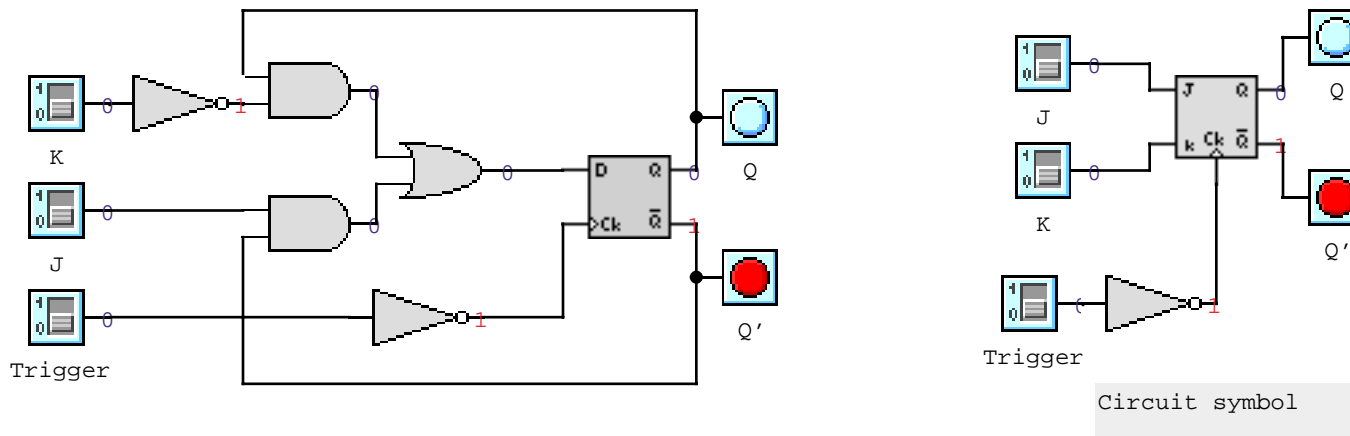
Just after clock returns to high; note transfer of state to second latch

## MASTER-SLAVE D FLIP-FLOP (3)



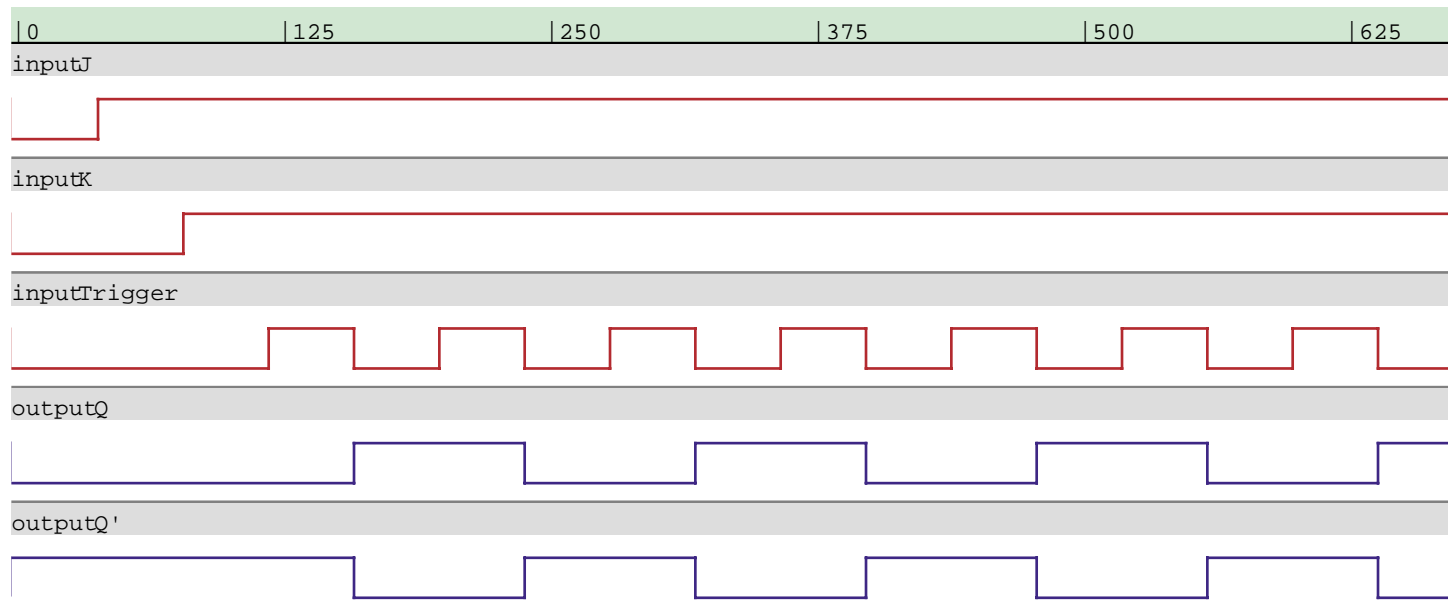
Timing diagram; note that state transitions occur on the leading (rising) edge of the clock signal

# PULSE-TRIGGERED JK FLIP-FLOP (1)



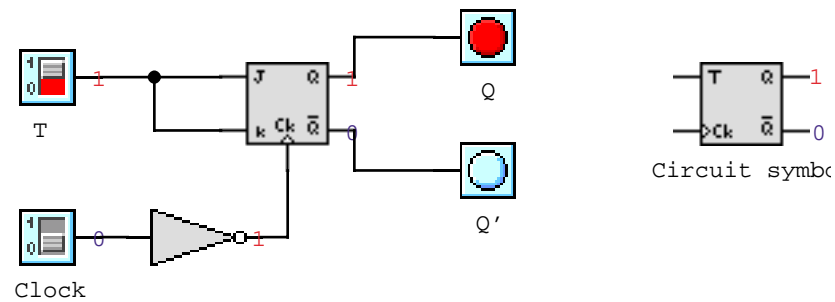
The JK flip-flop is similar to a master-slave SR flip-flop, except that when the J and K inputs are both high, the output “toggles” on the trailing (falling) edge of the clock signal. The circuit shown is equivalent to the circuit used in a commercial device, the SN7476.

# PULSE-TRIGGERED JK FLIP-FLOP (2)



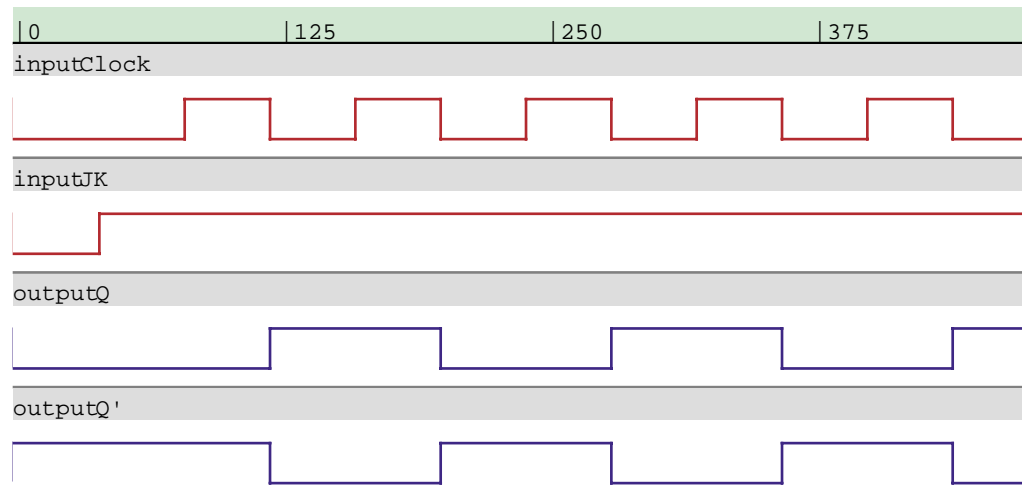
Timing diagram; note that state transitions occur on the trailing (falling) edge of the trigger signal

# T FLIP-FLOP (1)



T stands for “toggle”; note that this is simply a JK flip-flop with the J and K inputs held at  $V_{dd}$

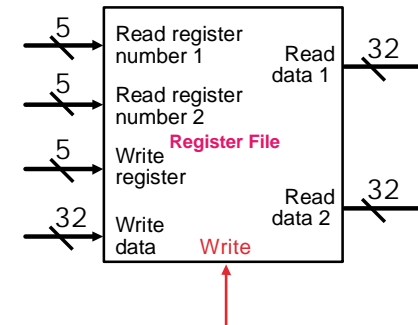
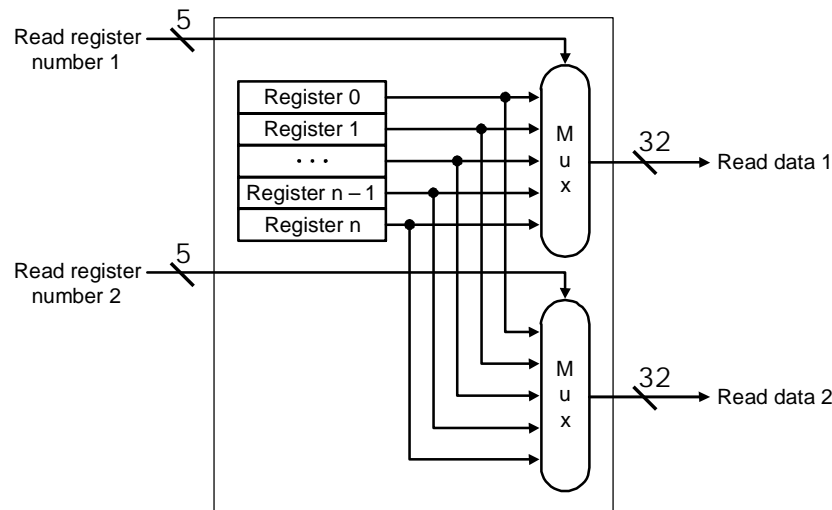
## T FLIP-FLOP (2)



Timing diagram; note that state transitions occur on the trailing (falling) edge of the clock signal



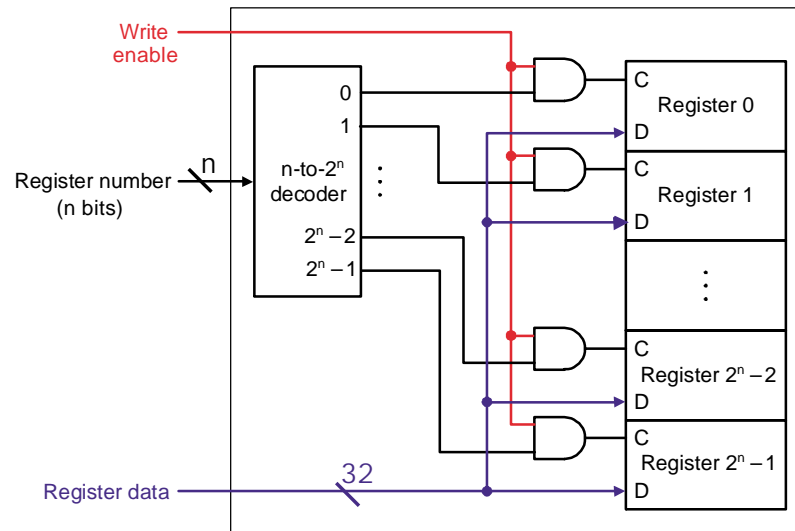
# REGISTER FILE



The register file is an array of arrays of flip-flops, addressed using a decoder, read using multiplexors. Data to be written to a particular register is **broadcast** to all registers, but only the specific register that is **selected** by an asserted “write enable” signal is modified.



## APPLICATION OF DECODER TO REGISTER ADDRESSING



Data is **broadcast** to all registers, but only the register **selected** by the decoder is modified

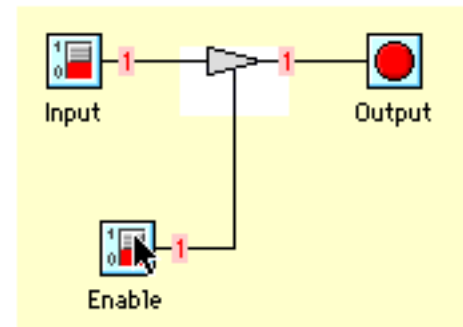
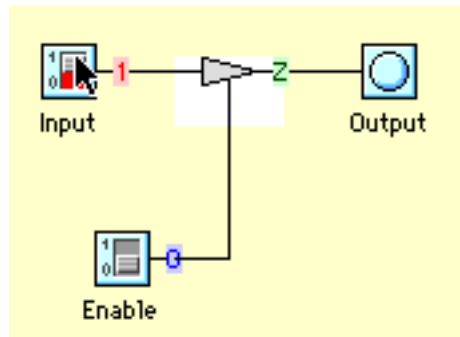
- Each register has an “enable” input (labeled C in the figure)
  - ▷ A register’s enable input must be asserted in order for data to be written to the register through the “data” input (labeled D)
  - ▷ The enable input is controlled by an AND gate
  - ▷ Both the signal from the decoder and the “write enable” signal must be asserted in order for the register’s enable input to be asserted

## BUFFERS

- A **buffer** is either a special-purpose memory or a type of amplifier
  - ▷ Special-purpose memory:
    - Useful when two systems have different clock frequencies or data rates
    - Accessed with input & output pointers to next locations for writing or reading data
    - May be implemented in hardware or software as an array, or a FIFO, or some other type of data structure
  - ▷ Buffer amplifier
    - May be used for power gain
    - Tristate buffer has a high-impedance output in the “off” state; used as transistor equivalent of a mechanical switch

## 3-STATE BUFFER

- A **3-state buffer** has 2 inputs (Data and Enable) and 1 output
  - ▷ Enable asserted: Output = input (state is either 0 or 1)
  - ▷ Enable deasserted: High-impedance state (denoted  $\times$  or **Z**)
    - Output can be driven by another device
  - ▷ Equivalent to a mechanical switch



**EXCITATION TABLE FOR 3-STATE BUFFER**

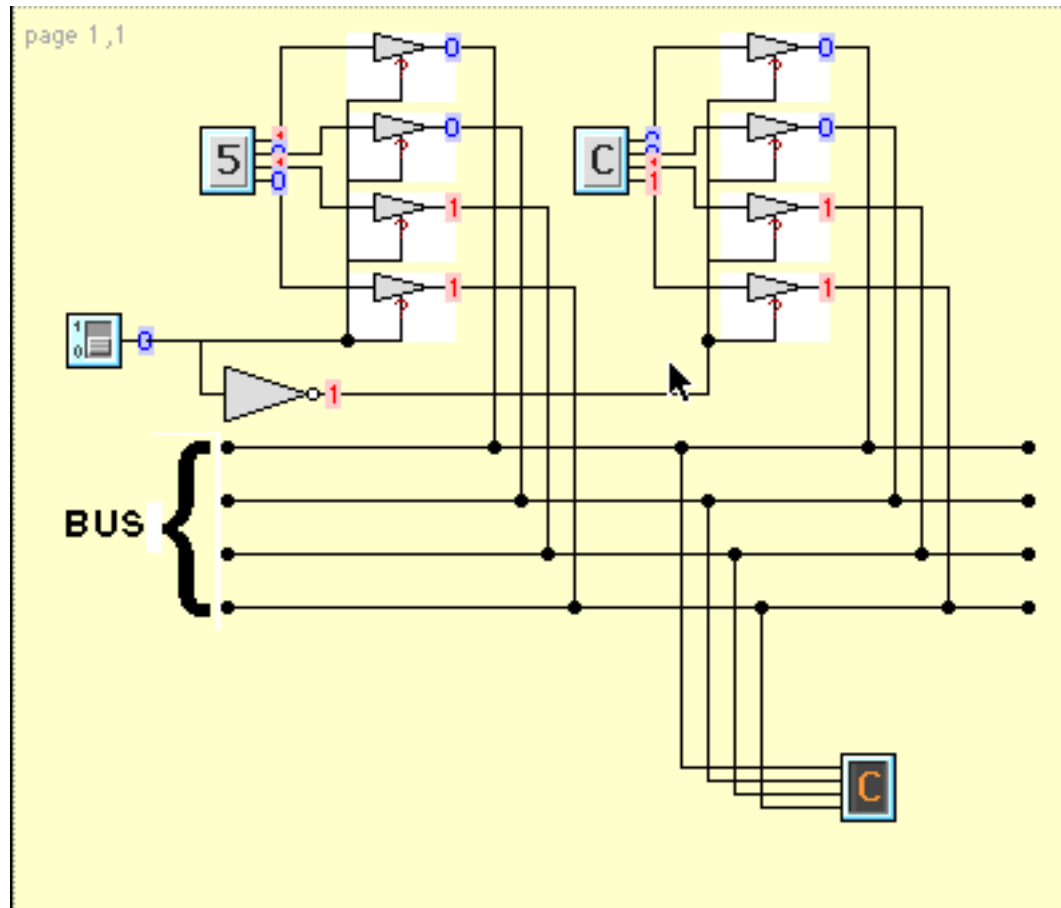
- A tristate buffer has 3 possible output values:
  - ▷ Asserted
  - ▷ Deasserted
  - ▷ High impedance (floating)

enable	in	out
0	0	Z
0	1	Z
1	0	0
1	1	1

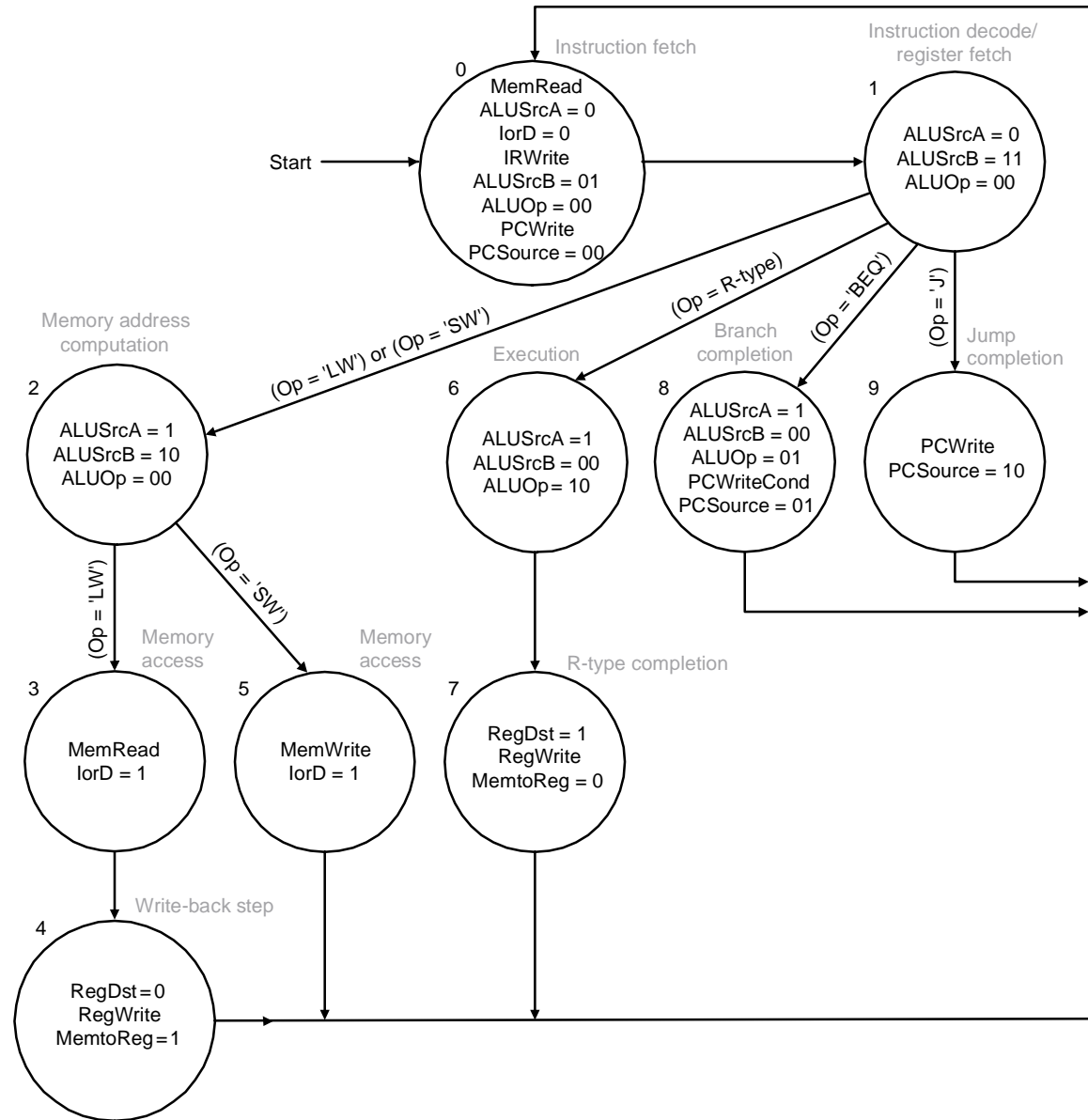
## BUSES

- **Bus:** A communication link shared by multiple subsystems
  - ▷ Physically: Parallel conductors (traces on die or PC board; cable)
  - ▷ Advantages:
    - Low cost (compared to point-to-point wiring)
    - Versatility of interconnections
  - ▷ Disadvantages:
    - Possible communication bottleneck
    - Shared resource  $\Rightarrow$  contention
  - ▷ Organization:
    - Control lines to signal & acknowledge requests
    - Data lines to carry addresses, data or commands

# USE OF TRISTATES TO ENABLE/DISABLE BUS ACCESS



# MIPS FSM



How many bits are needed to index the states?

**ROMs (1)**

- A **read-only memory (ROM)** is a combinational logic circuit such that:
  - ▷ There are  $n$  **address inputs** followed by an  $n$ -to- $2^n$  address decoder
  - ▷ Each address labels a unique  $m$ -bit word
  - ▷ ROM capacity =  $2^n \times m$  bits
  - ▷ Used to store:
    - Compiled programs that must always be available, such as are used in embedded systems
      - ◊ Printers
      - ◊ Modems
    - Microinstructions to determine values of control signals in CISC processors