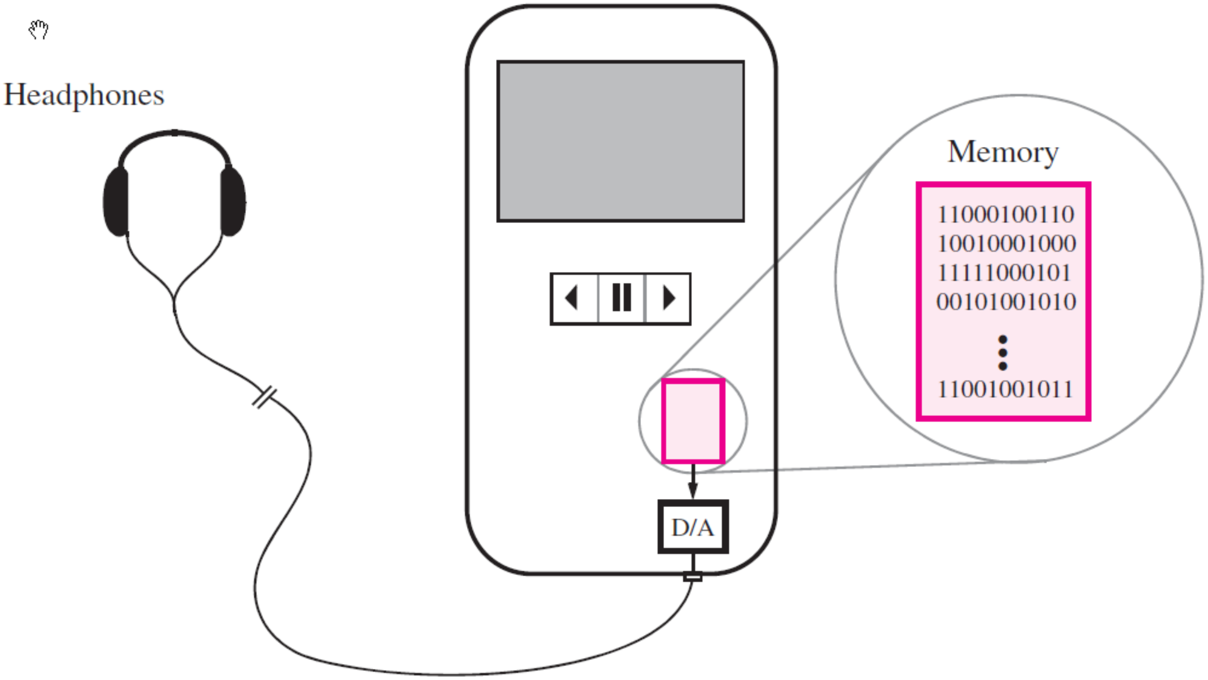
# Introduction & Motivation

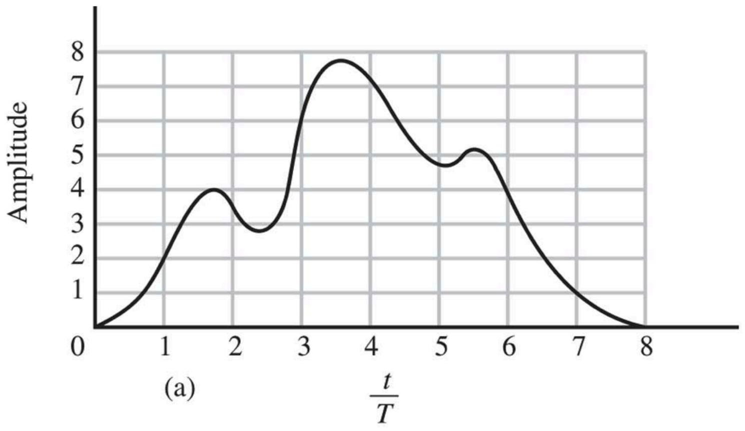
CPEN 230 – Introduction to Digital Logic

# Digital Abstraction

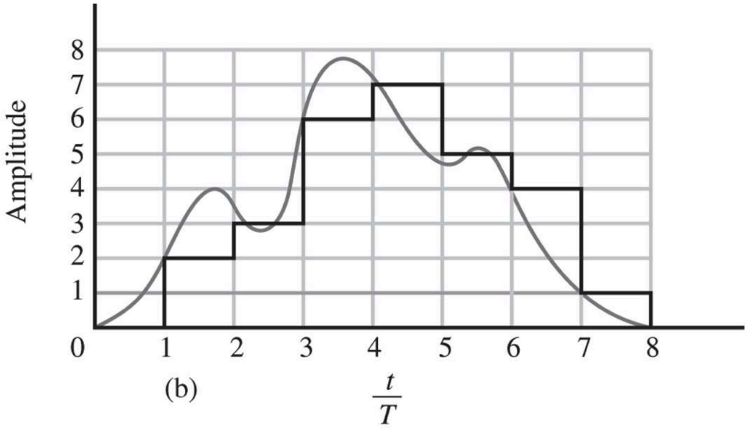
Abstraction = hiding details when they are not important
Abstraction helps to manage complexity

- The world is “digital” with analog at the edges





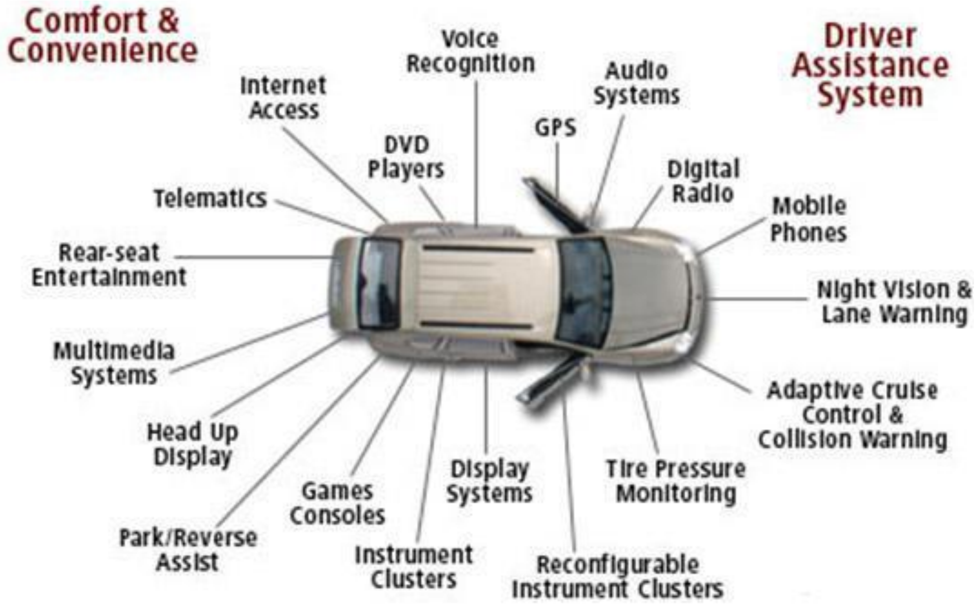
(a) analog signal

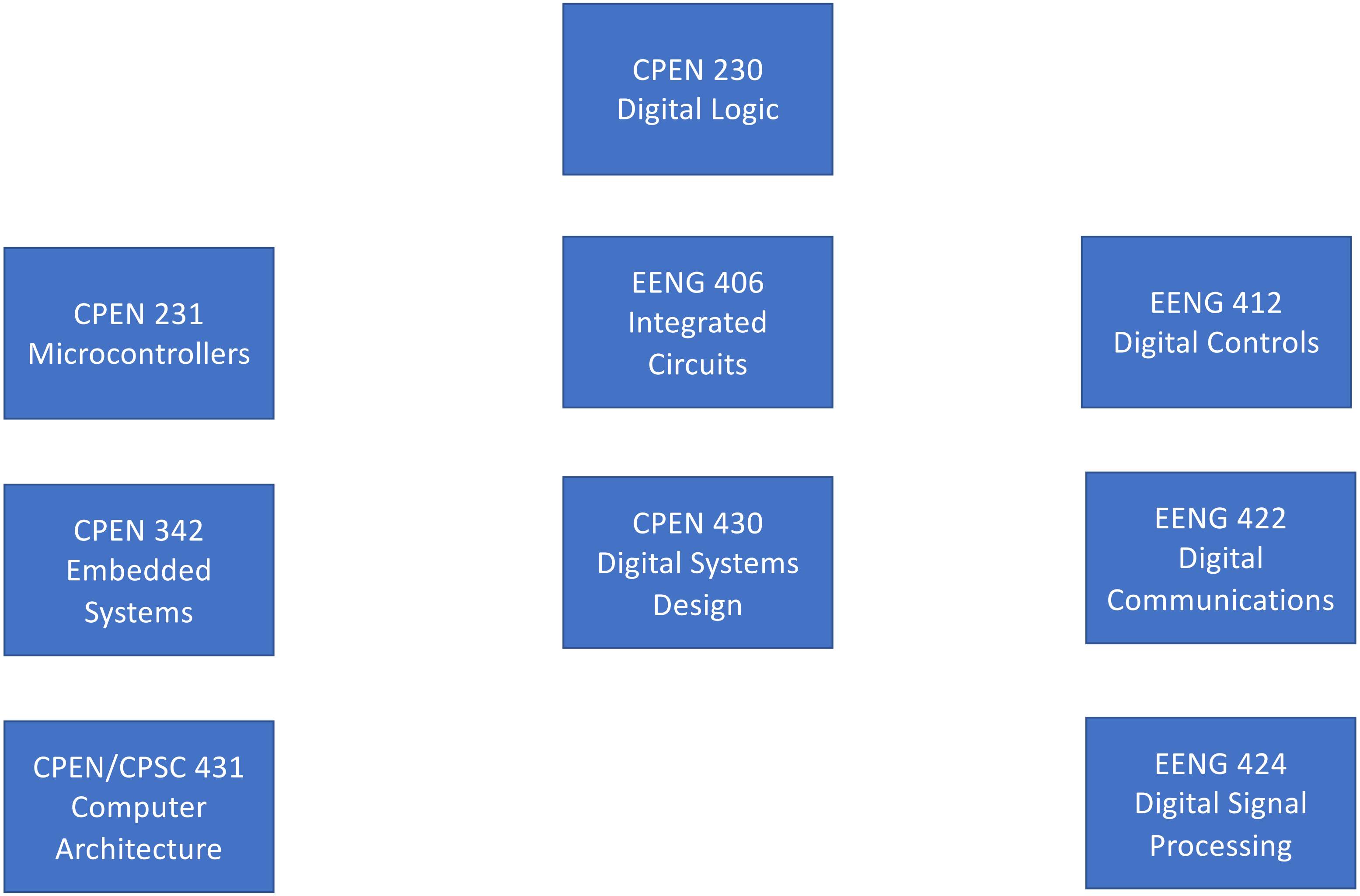


(b) digital signal

# Common Applications

- Desktop Computers
- Notebooks
- Smartphones
- Embedded Systems





CPEN 230
Digital Logic

CPEN 231
Microcontrollers

CPEN 342
Embedded Systems

CPEN/CPSC 431
Computer Architecture

EENG 406
Integrated Circuits

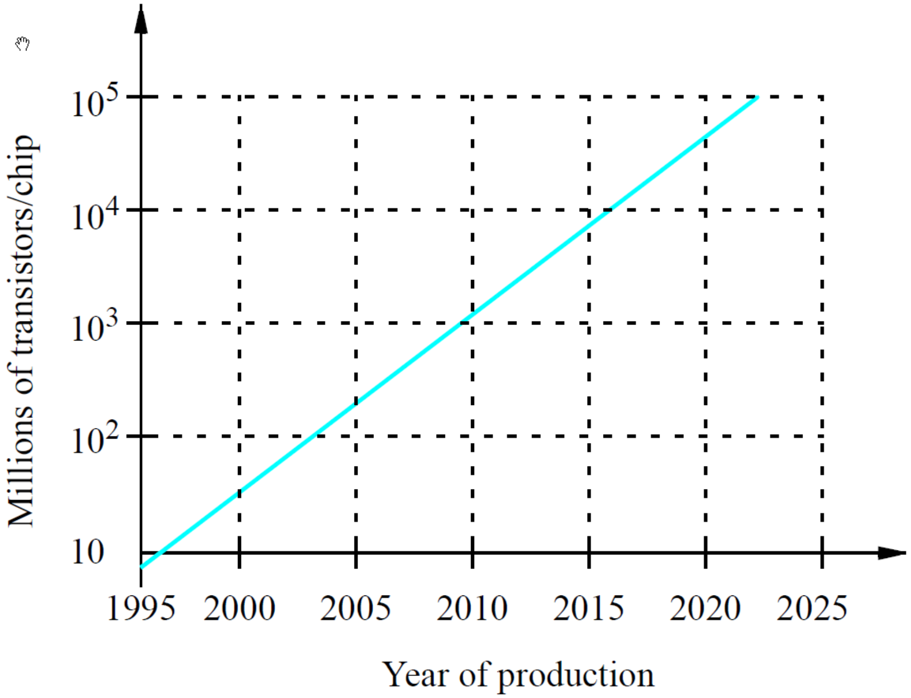
CPEN 430
Digital Systems Design

EENG 412
Digital Controls

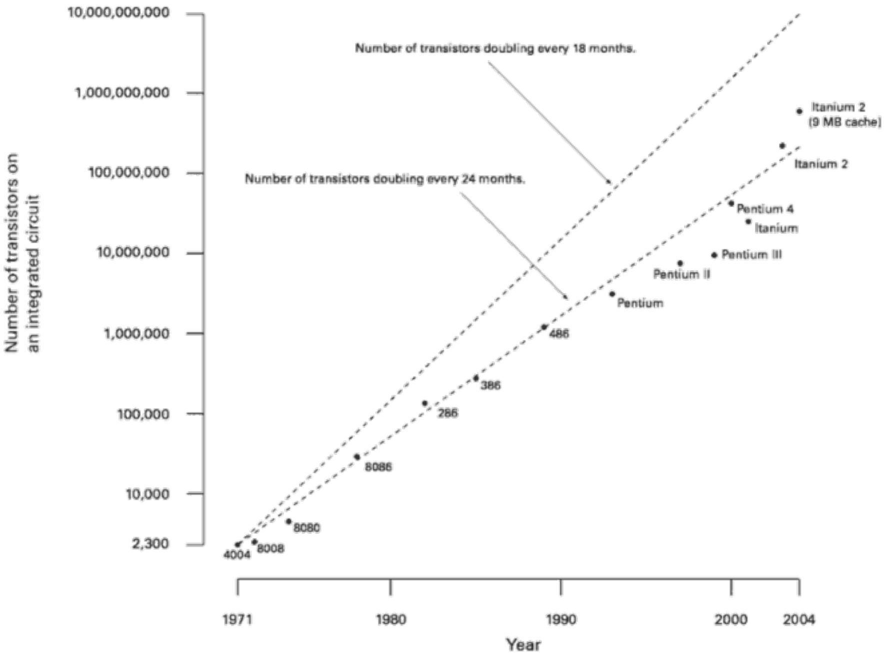
EENG 422
Digital Communications

EENG 424
Digital Signal Processing

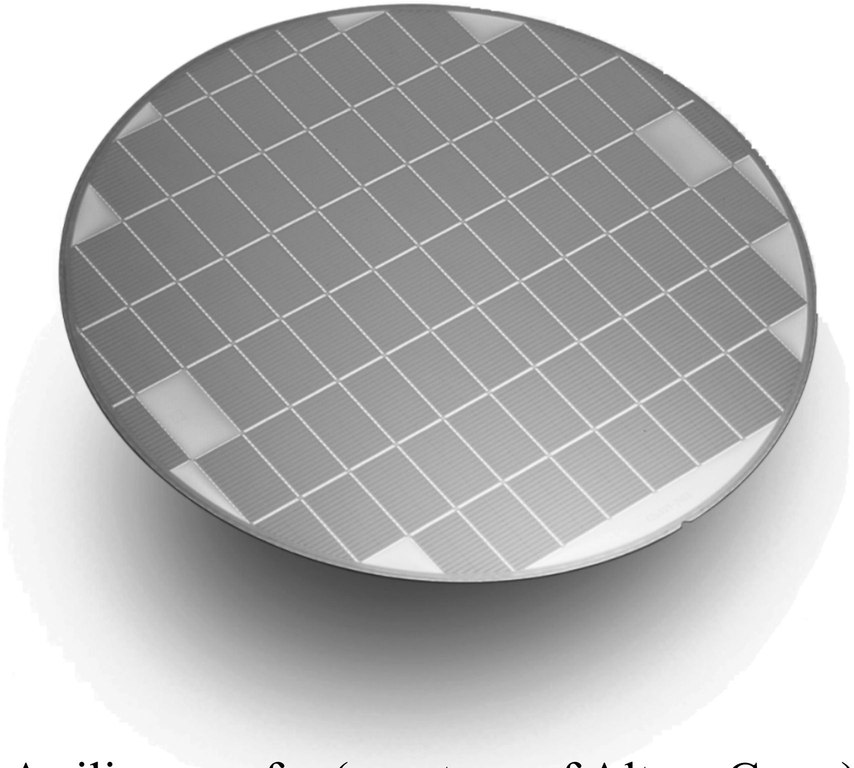
# Moore’s Law



An estimate of the maximum number of transistors per chip over time.



# Enabling Technology



A silicon wafer (courtesy of Altera Corp.).

- The focus of this class is on digital circuits: i.e. the interconnection of logic gates

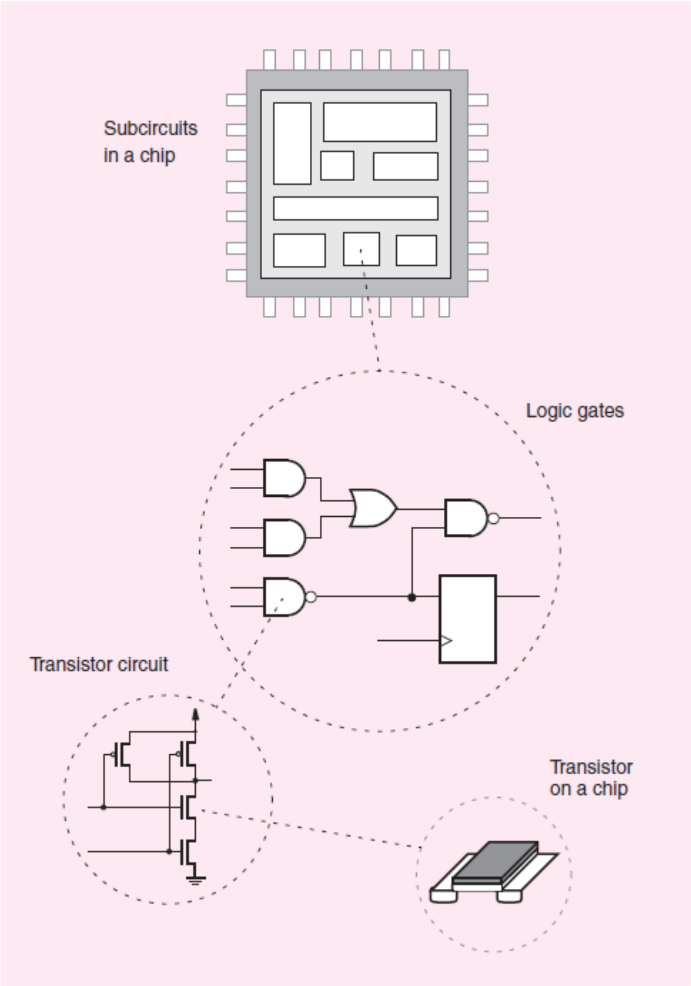
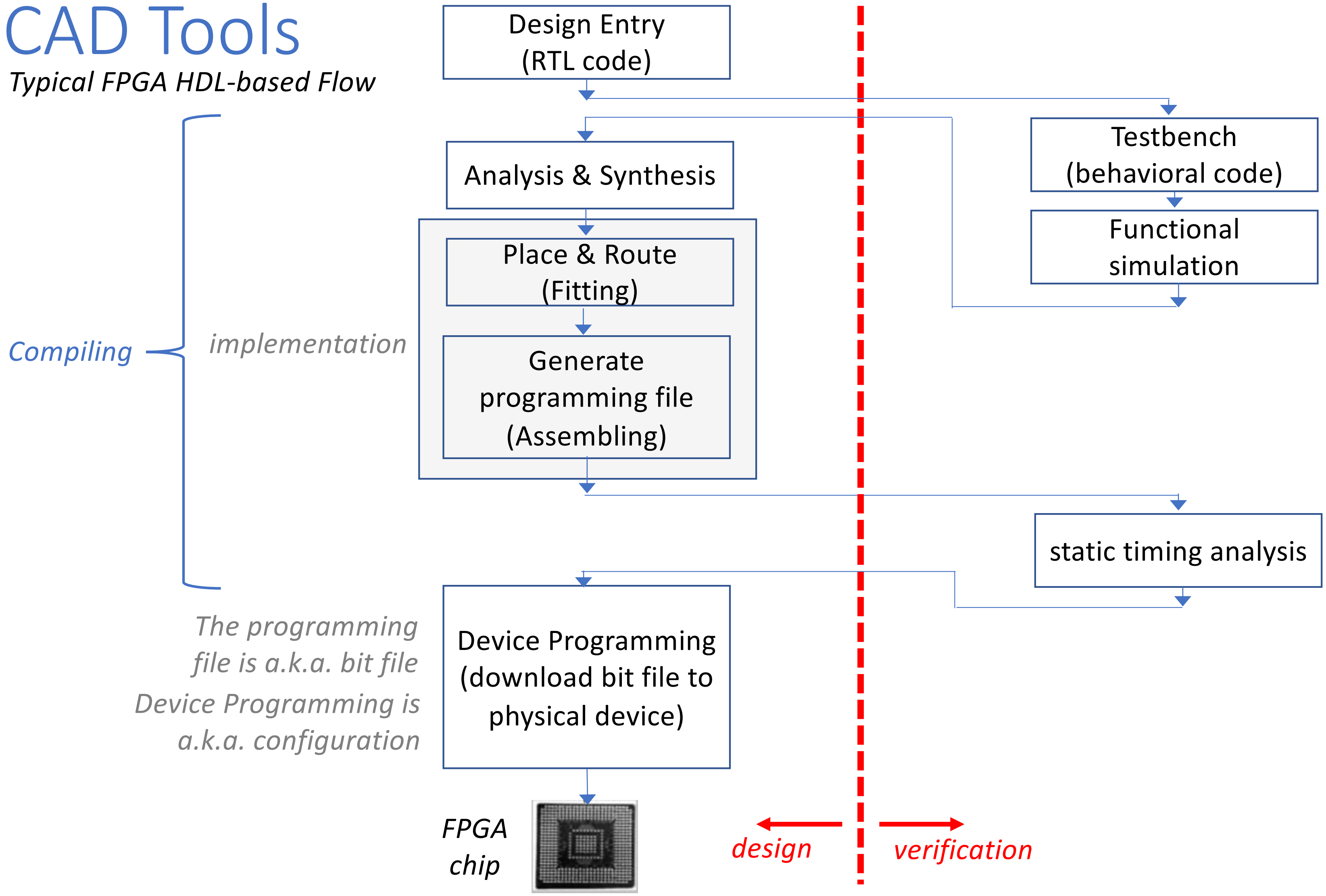


**Figure 1.4** A digital hardware system (Part *b*).

An example of chip (CPU)

# CAD Tools

*Typical FPGA HDL-based Flow*



Design Entry (RTL code)

Analysis & Synthesis

Place & Route (Fitting)

Generate programming file (Assembling)

*Compiling*

*implementation*

Testbench (behavioral code)

Functional simulation

static timing analysis

*The programming file is a.k.a. bit file*

*Device Programming is a.k.a. configuration*

Device Programming (download bit file to physical device)

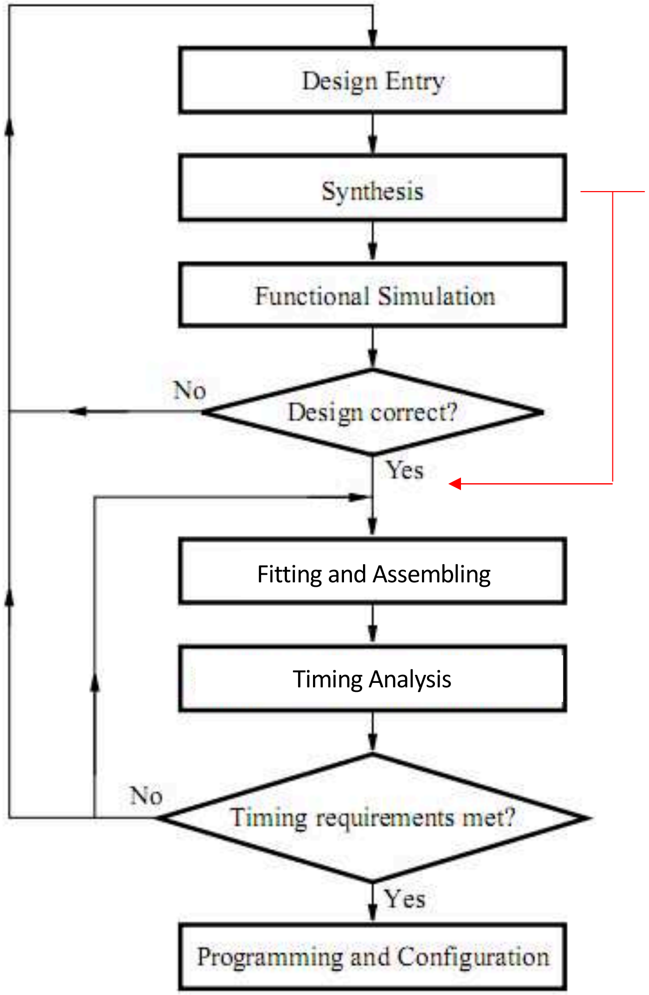
*FPGA chip*

*design*

*verification*

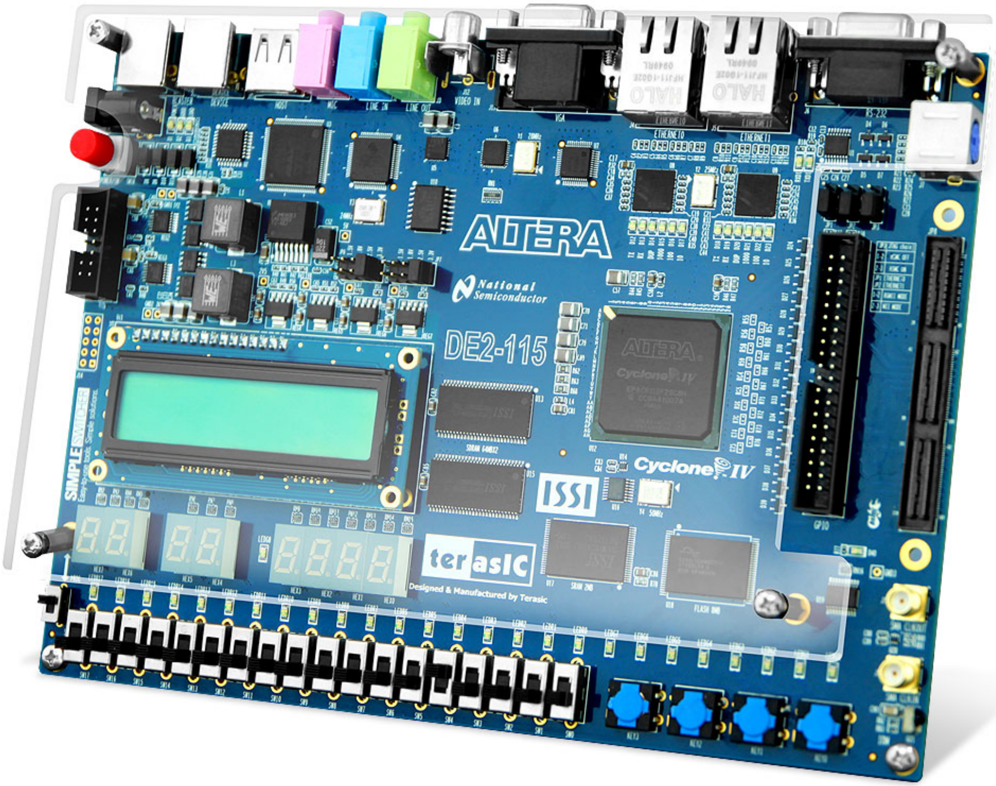
# CAD Tools

*The flow is iterative!*



*Unless you don't You don't trust your HDL coding skills you should first check that the functional simulation is correct and later run the synthesis*

Figure 1.Typical CAD flow.



An FPGA (Field Programmable Gate Array) board.

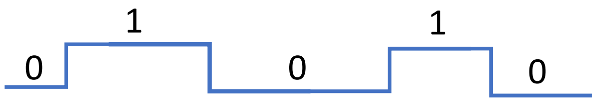
# Digital Abstraction

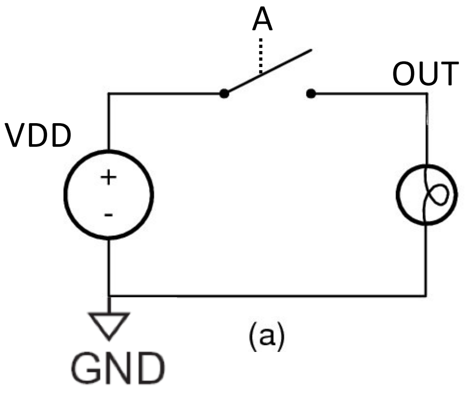
Key idea:

assume we have only two voltage levels

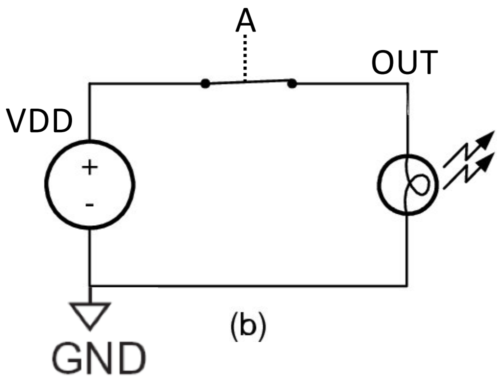
(i.e., assume binary signals)

- VDD <--> 1 <--> H
- GND <--> 0 <--> L





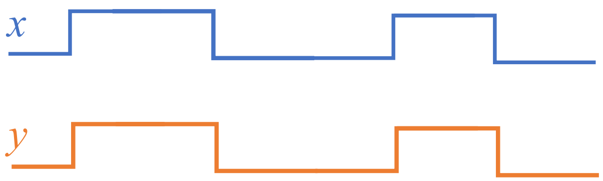
(a)



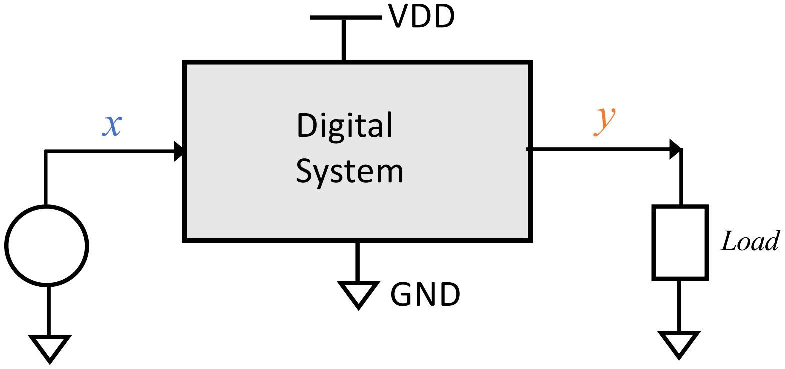
(b)

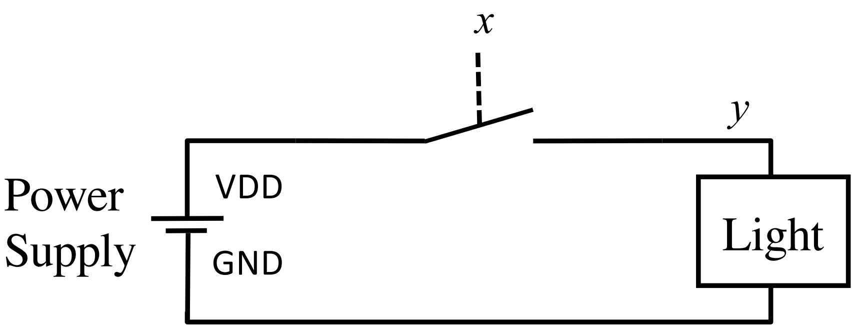
# Binary Waveforms

Assume we have a switch that when the control signal x is at VDD (x=1) the switch closes and when x is at GND (x=0) the switch opens

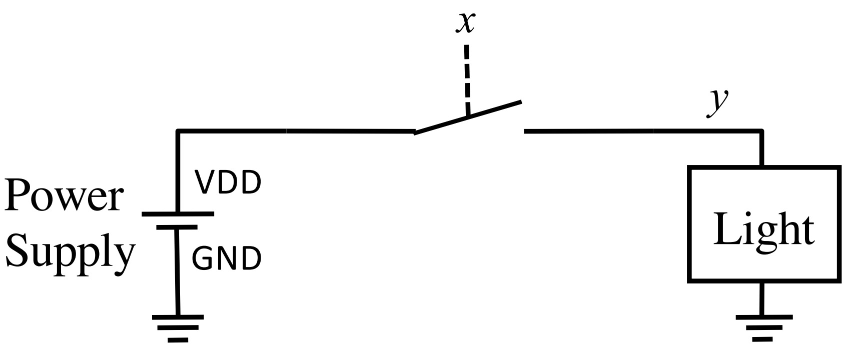


*x = input variable* *y = output variable*



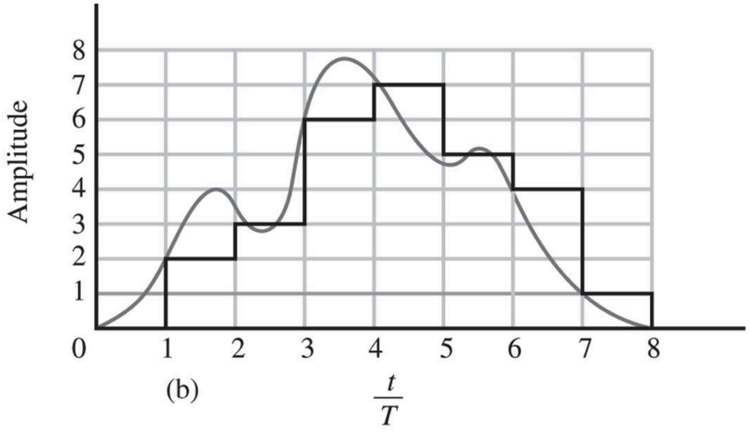


(a) Simple connection to a battery

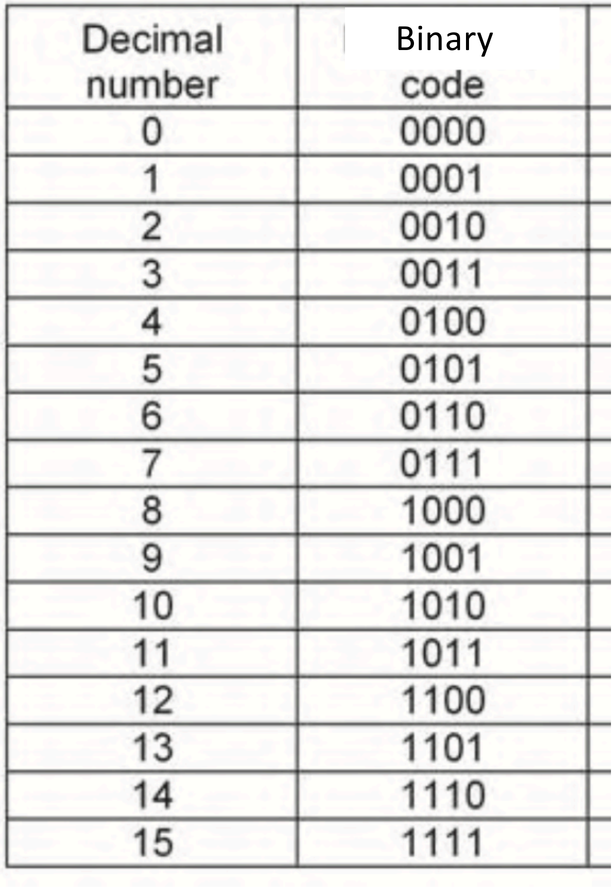


(b) Using a ground connection as the return path

# Binary digits (bits)



If we “slice” the amplitude of our analog signal in 8 values, we need 3 binary signals (i.e. 3 bits) to represent it ( 8 = $2^3$ ).



| Decimal number | Binary code |
|---|---|
| 0 | 0000 |
| 1 | 0001 |
| 2 | 0010 |
| 3 | 0011 |
| 4 | 0100 |
| 5 | 0101 |
| 6 | 0110 |
| 7 | 0111 |
| 8 | 1000 |
| 9 | 1001 |
| 10 | 1010 |
| 11 | 1011 |
| 12 | 1100 |
| 13 | 1101 |
| 14 | 1110 |
| 15 | 1111 |

# Number Systems

$$num = \sum_{k=0}^{N-1} d_k R^k$$

## Decimal numbers

1000's column
100's column
10's column
1's column

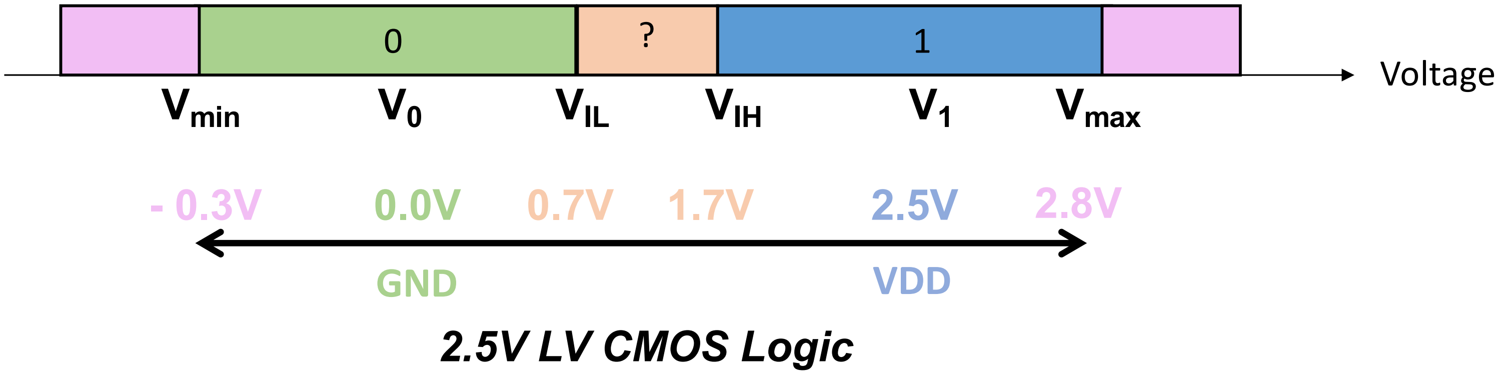
$$5374_{10} = \underbrace{5 \times 10^3}_{\text{five thousands}} + \underbrace{3 \times 10^2}_{\text{three hundreds}} + \underbrace{7 \times 10^1}_{\text{seven tens}} + \underbrace{4 \times 10^0}_{\text{four ones}}$$

## Binary numbers

8's column
4's column
2's column
1's column

$$1101_2 = \underbrace{1 \times 2^3}_{\text{one eight}} + \underbrace{1 \times 2^2}_{\text{one four}} + \underbrace{0 \times 2^1}_{\text{no two}} + \underbrace{1 \times 2^0}_{\text{one one}} = 13_{10}$$

# Digital Abstraction:
# Voltage ranges of binary signals



***2.5V LV CMOS Logic***

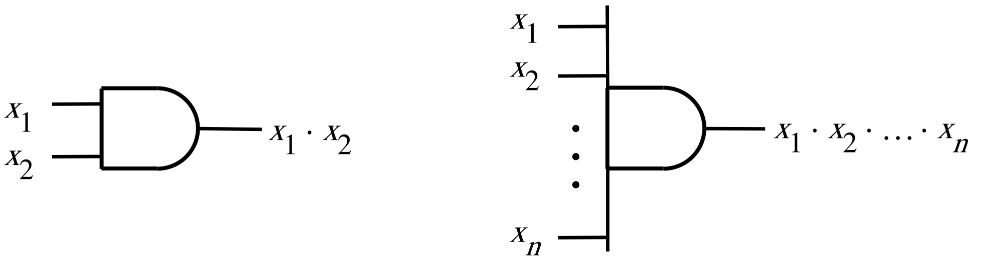
The building blocks of a digital circuit are called logic gates

$AND = \cdot = \vee = \cap = \&$

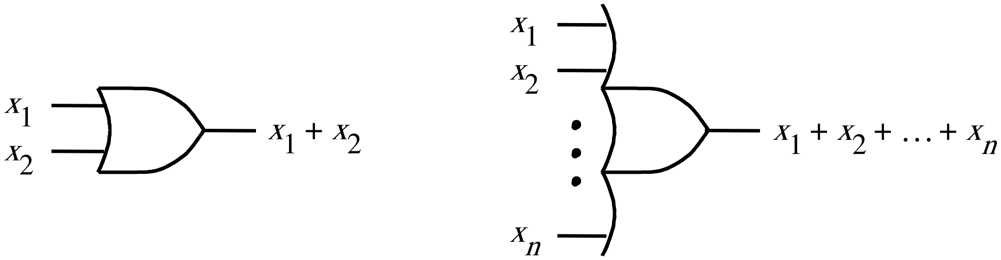
$OR = \ + = \vee = \cup = |$

$NOT = \quad ' = \ \bar{} \ = \sim = !$

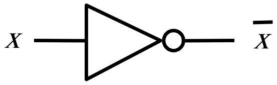
The three Basic Logic gates



(a) AND gates



(b) OR gates

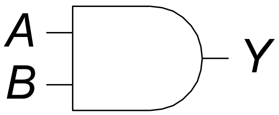


(c) NOT gate

A digital circuit is a "network" of logic gates (i.e. a bunch of logic gates wired together)

# The Basic Logic Gates: AND, OR, NOT

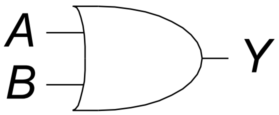
## AND



$Y = AB$

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

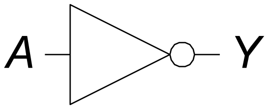
## OR



$Y = A + B$

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

## NOT

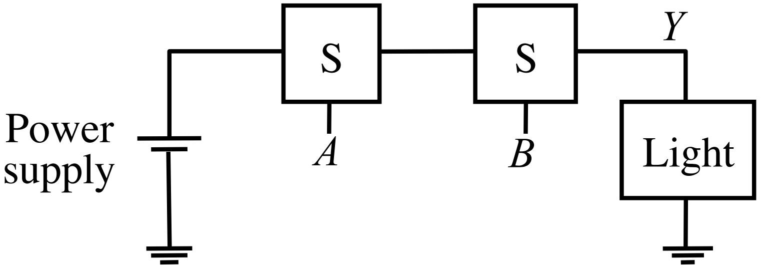


$Y = \overline{A}$

| A | Y |
|---|---|
| 0 | 1 |
| 1 | 0 |

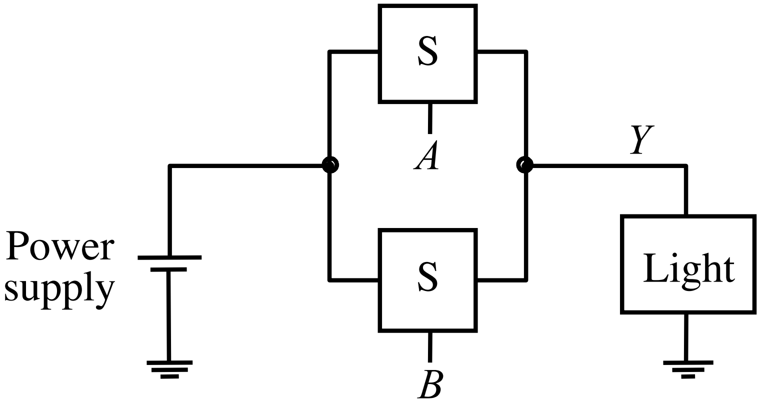
These three basic components are all you need to build any possible logic circuit you want

# AND, OR, NOT



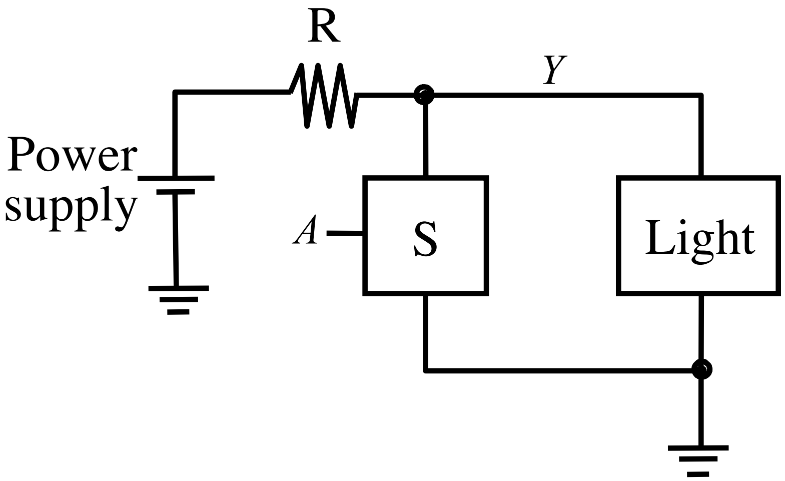
| *A* | *B* | *Y* |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

(a) The logical AND function (series connection)



| *A* | *B* | *Y* |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

(b) The logical OR function (parallel connection)

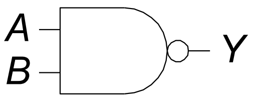


(c) The logical NOT function

| *A* | *Y* |
|---|---|
| 0 | 1 |
| 1 | 0 |

# More Logic Gates: NAND, NOR, and BUFFER (inverting AND, OR, and NOT)

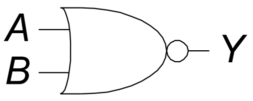
**NAND**



$Y = \overline{AB}$

| $A$ | $B$ | $Y$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

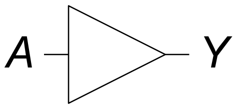
**NOR**



$Y = \overline{A + B}$

| $A$ | $B$ | $Y$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

**BUF**



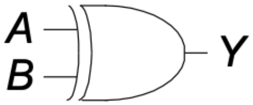
$Y = A$

| $A$ | $Y$ |
|---|---|
| 0 | 0 |
| 1 | 1 |

It turns out that instead of using the three basic gates (AND, OR, NOT) you can build any logic circuit you want also by using only NAND gates or only NOR gates

# More Logic Gates: XOR, XNOR

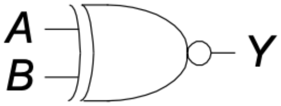
## XOR



$Y = A \oplus B$

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

## XNOR

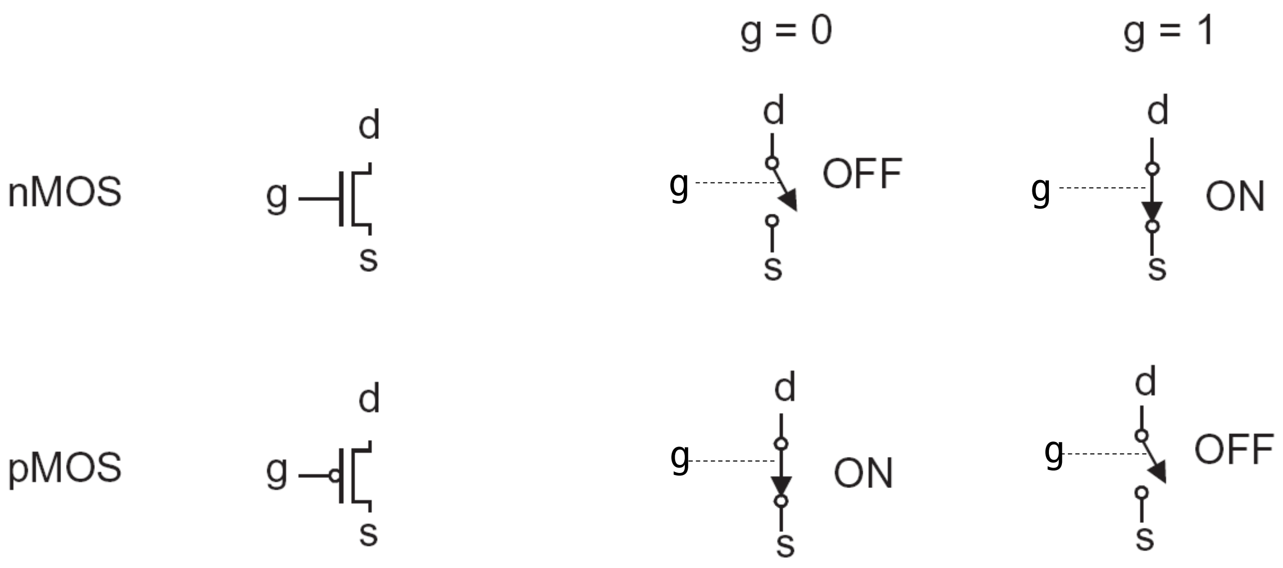


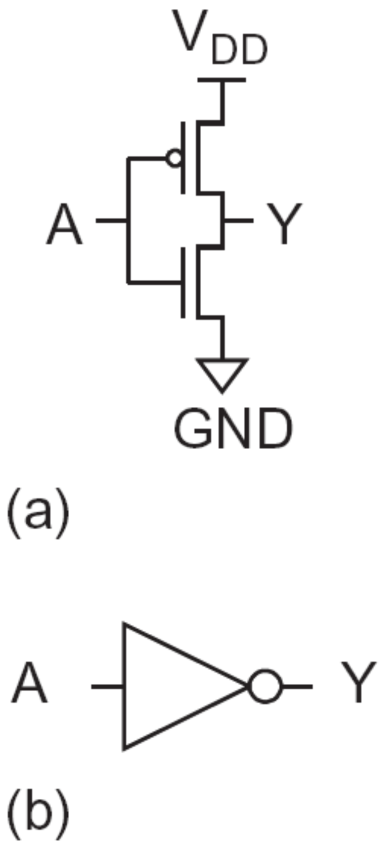
$Y = \overline{A \oplus B}$

| A | B | Y |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

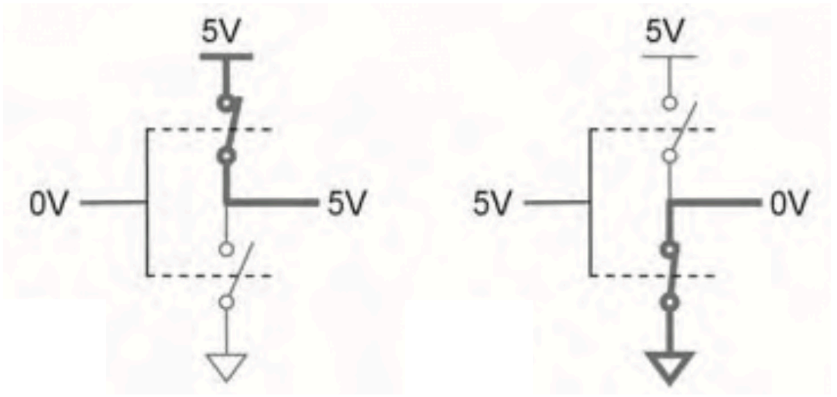
# Beneath the Digital Abstraction

*… there are transistors*





Example: VDD = 5V, GND = 0V



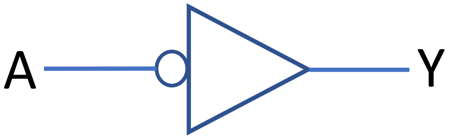


**FIGURE 1.11**
Inverter schematic
(a) and symbol
(b) $Y = \overline{A}$

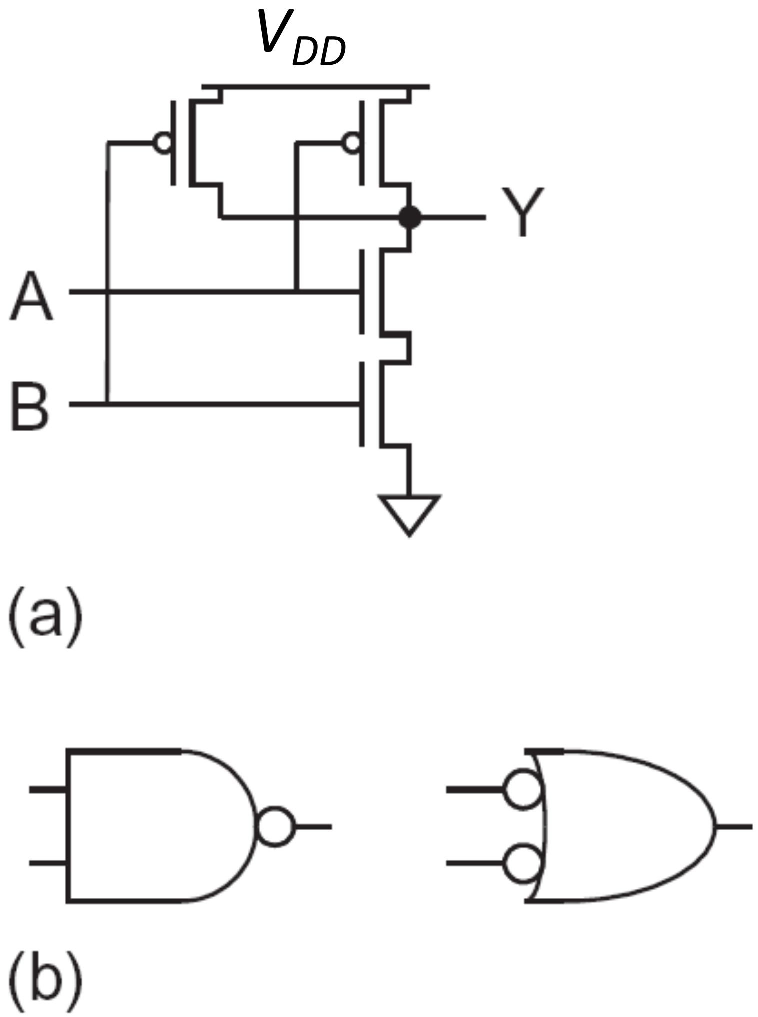


**FIGURE 1.12** 2-input NAND gate schematic (a) and symbol (b) $Y = \overline{A \cdot B}$

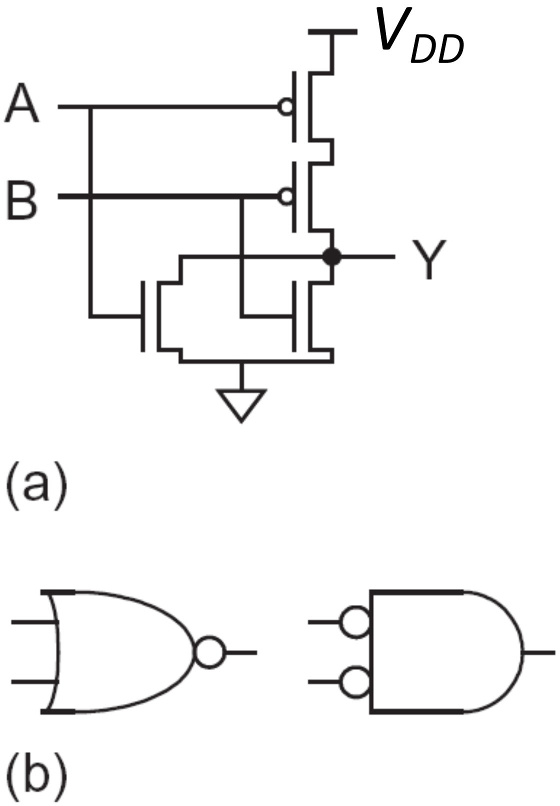
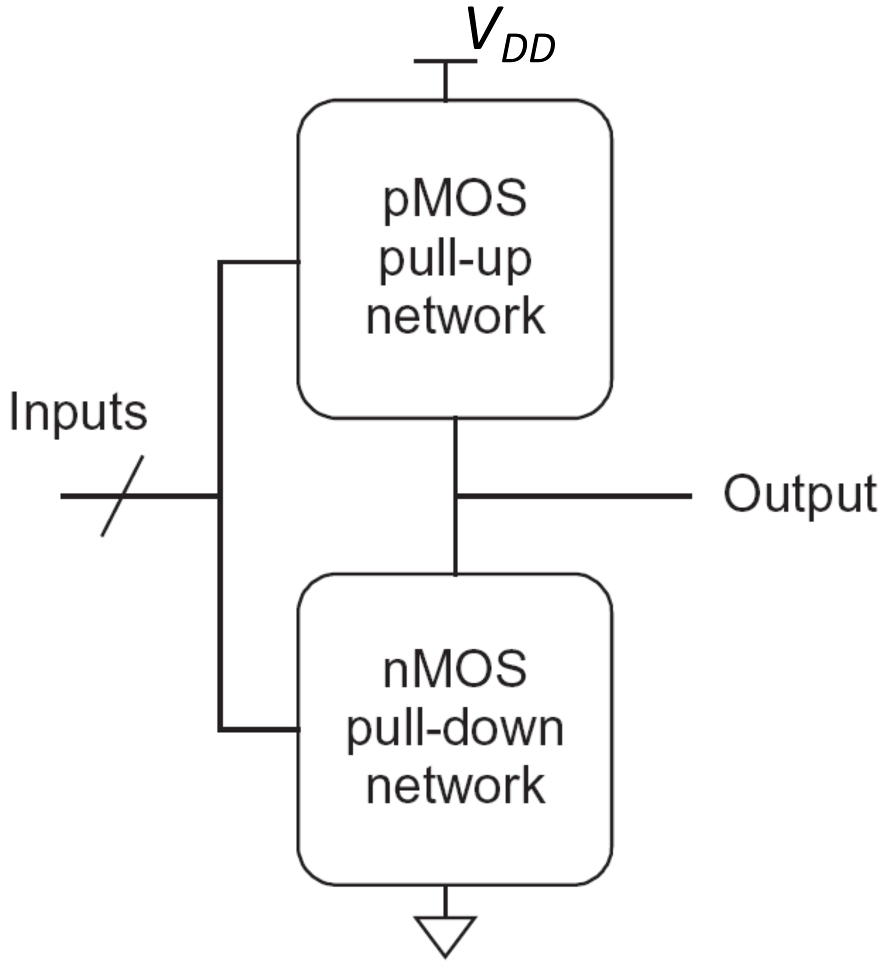
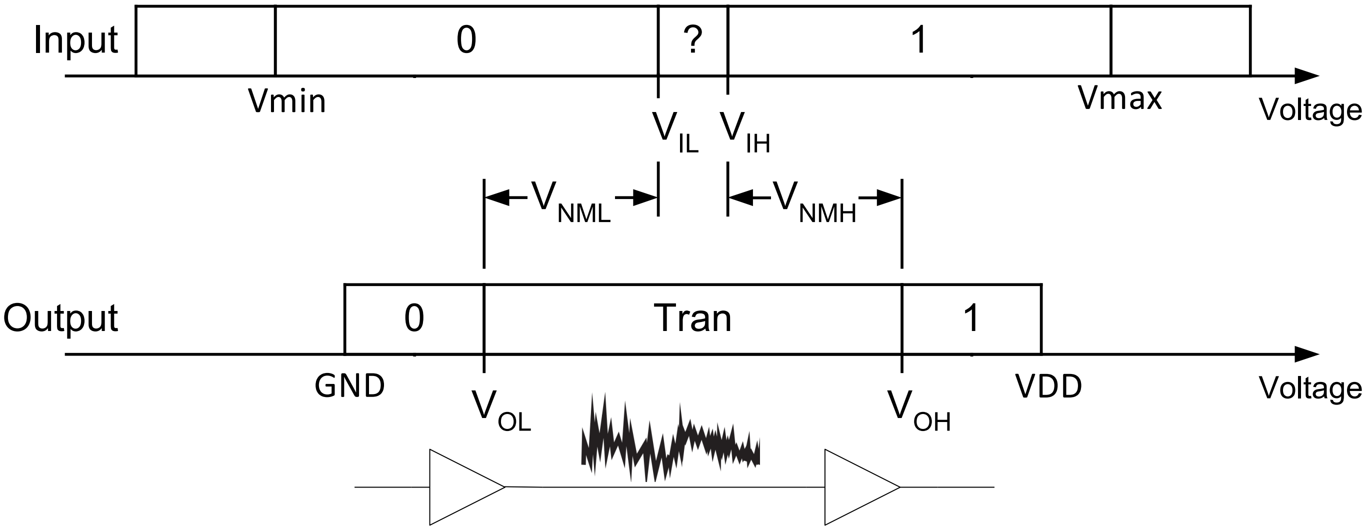


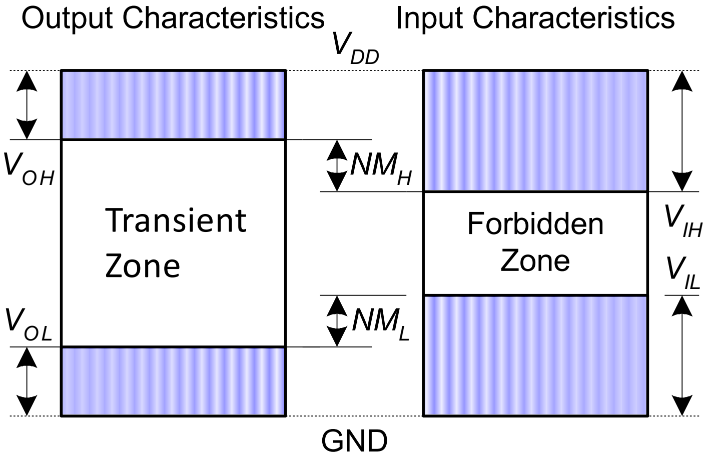
**FIGURE 1.16** 2-input NOR gate schematic (a) and symbol (b) $Y = \overline{A + B}$

# General CMOS Logic Gate



# Logic Levels and Noise Margins

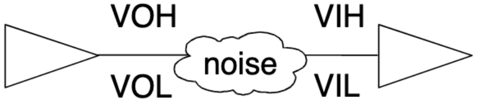


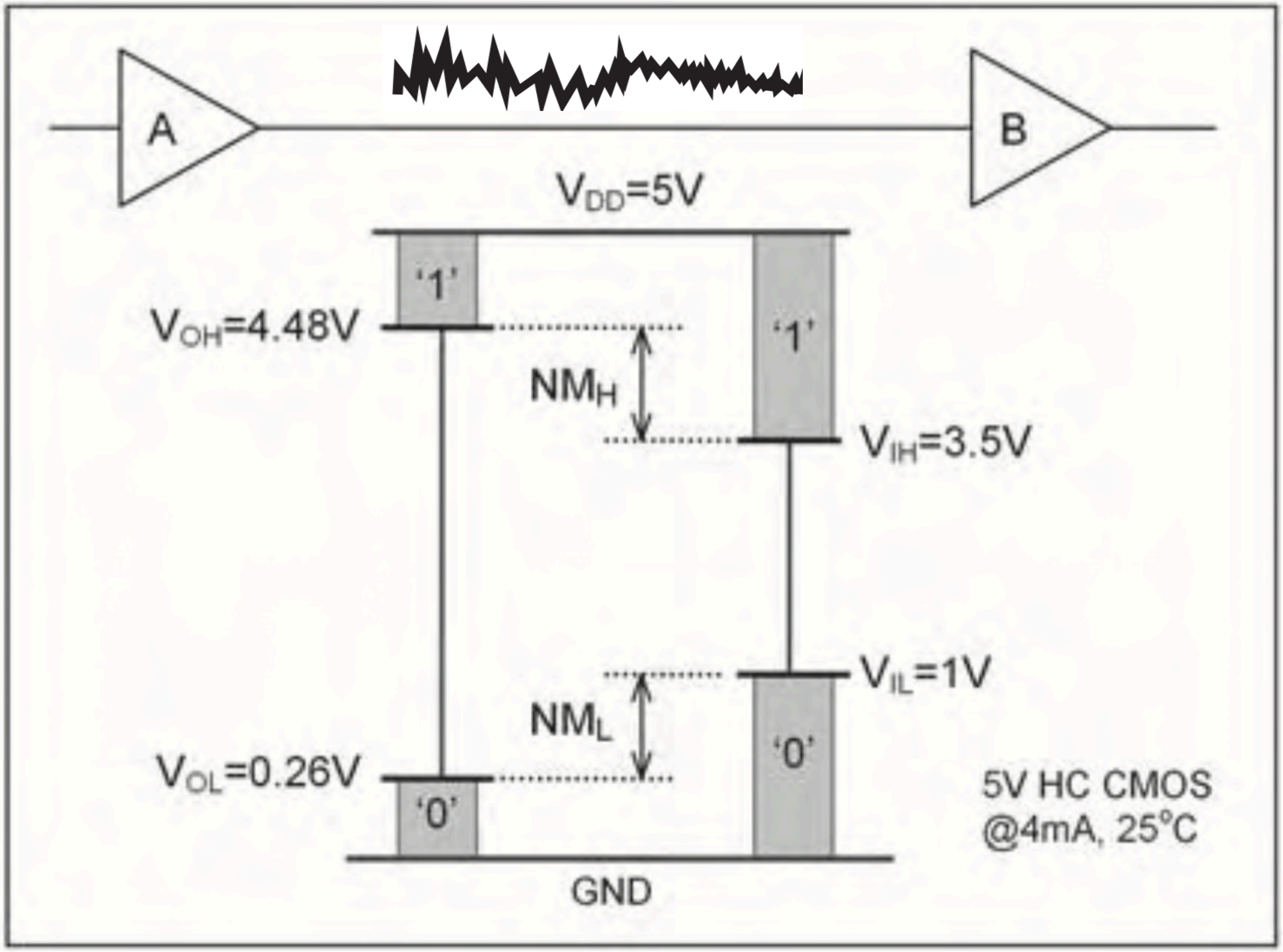


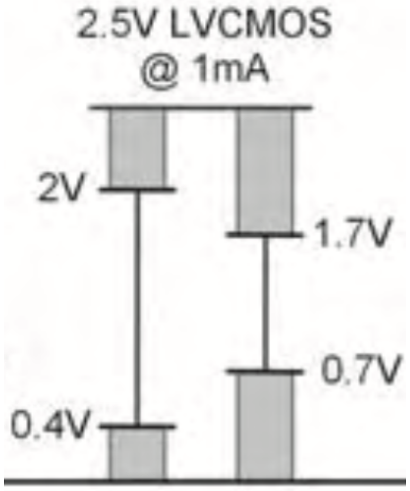
*Digital gates are restoring*

if Noise added to Input < Noise Margin

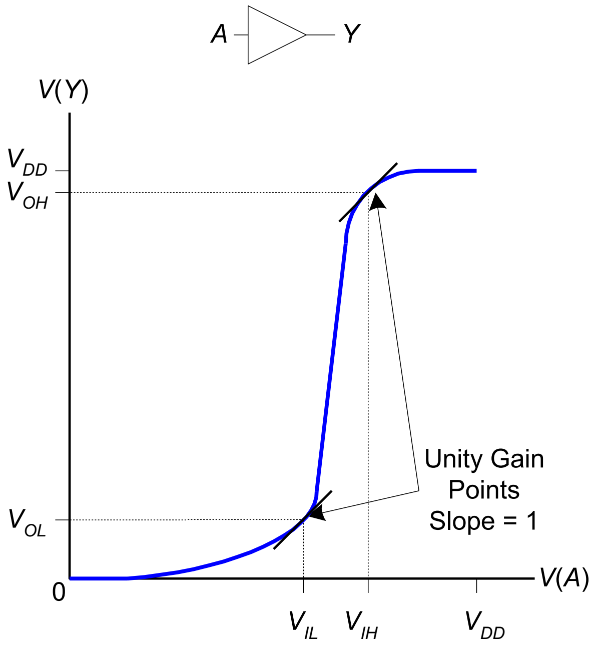
we get the correct value at output

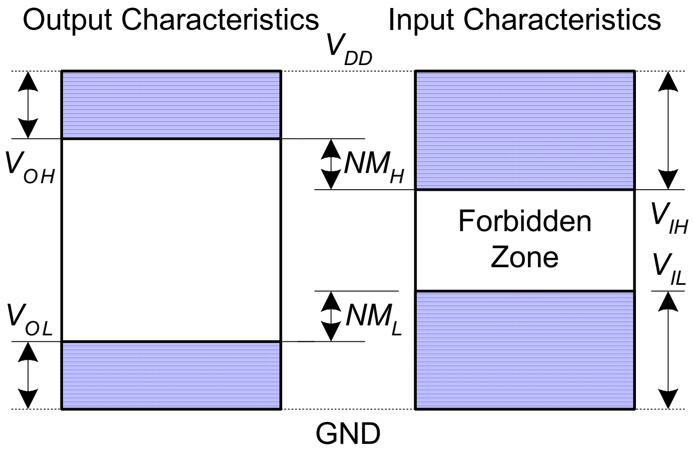



A
B
$V_{DD}$=5V
'1'
$V_{OH}$=4.48V
$NM_H$
'1'
$V_{IH}$=3.5V
$V_{IL}$=1V
$NM_L$
'0'
$V_{OL}$=0.26V
'0'
5V HC CMOS
@4mA, 25°C
GND


2.5V LVCMOS
@ 1mA
2V
1.7V
0.7V
0.4V

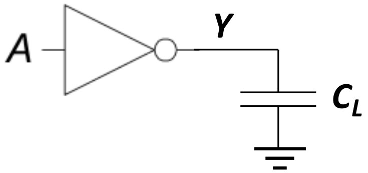
# DC Transfer Characteristics





If gain < 1 the noise superimposed to a given level gets “attenuated”

# VDD Scaling



- In 1970’s and 1980’s, $V_{DD}$ = 5 V

- $V_{DD}$ has dropped
  - Save power

$$P_d = \alpha \cdot C_L \cdot f_{clk} \cdot V_{DD}^2$$

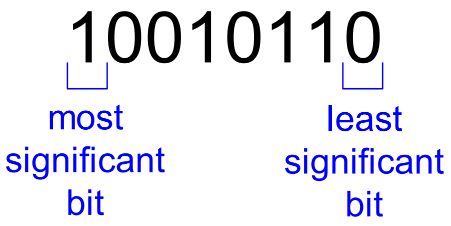
- 3.3 V, 2.5 V, 1.8 V, 1.5 V, 1.2 V, 1.0 V, …
  - Be careful connecting chips with different supply voltages

# Logic Family Examples

| Logic Family | $V_{DD}$ | $V_{IL}$ | $V_{IH}$ | $V_{OL}$ | $V_{OH}$ |
|---|---|---|---|---|---|
| TTL | 5 (4.75 - 5.25) | 0.8 | 2.0 | 0.4 | 2.4 |
| CMOS | 5 (4.5 - 6) | 1.35 | 3.15 | 0.33 | 3.84 |
| LVTTL | 3.3 (3 - 3.6) | 0.8 | 2.0 | 0.4 | 2.4 |
| LVCMOS | 3.3 (3 - 3.6) | 0.9 | 1.8 | 0.36 | 2.7 |

# Bits, Bytes, Nibbles, Words, …

Bits

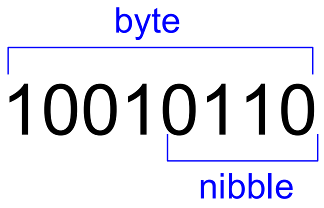


10010110

most significant bit

least significant bit

Bytes & Nibbles

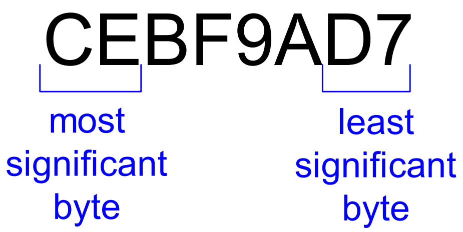


byte

10010110

nibble

Bytes



CEBF9AD7

most significant byte

least significant byte

??

# Hexadecimal Numbers

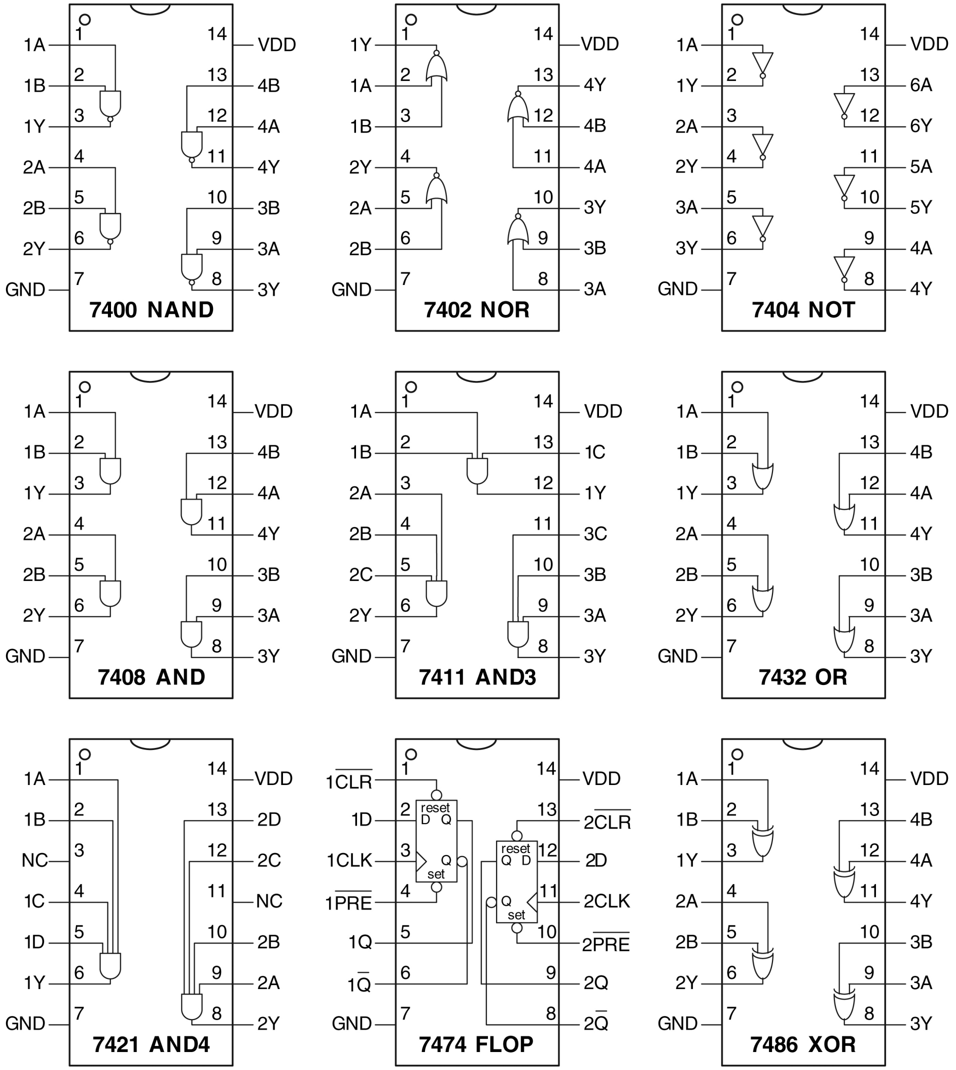
| Hex Digit | Decimal Equivalent | Binary Equivalent |
|---|---|---|
| 0 | 0 | 0000 |
| 1 | 1 | 0001 |
| 2 | 2 | 0010 |
| 3 | 3 | 0011 |
| 4 | 4 | 0100 |
| 5 | 5 | 0101 |
| 6 | 6 | 0110 |
| 7 | 7 | 0111 |
| 8 | 8 | 1000 |
| 9 | 9 | 1001 |
| A | 10 | 1010 |
| B | 11 | 1011 |
| C | 12 | 1100 |
| D | 13 | 1101 |
| E | 14 | 1110 |
| F | 15 | 1111 |

# Next Time

- Types of logic circuits:
  combinational circuits vs. sequential circuits
- Types of Integrated Circuits (ICs)
- Boolean Algebra

Combinational --> NOT Combinatorial

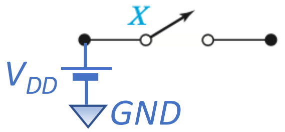
# Common 74xx-series logic gates



# Boolean Algebra

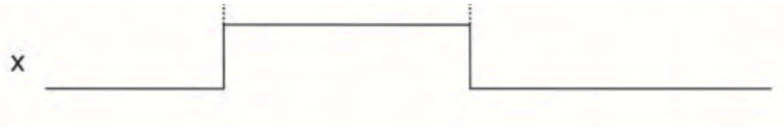
CPEN 230 – Introduction to Digital Logic

# Review

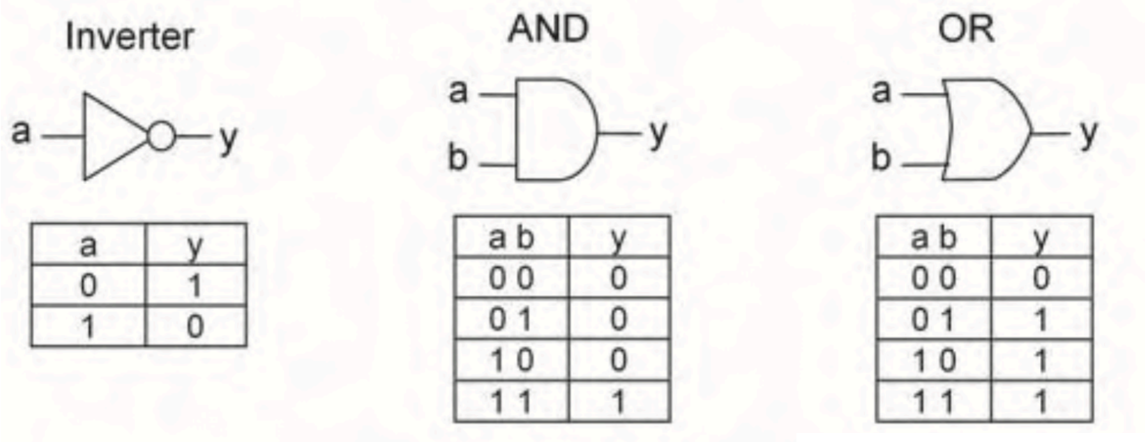


$X = 0 \rightarrow$ switch open
$X = 1 \rightarrow$ switch closed

- Binary (or Boolean or Digital or Logic) Signal (or Variable)



- Fundamental Logic Functions (or Operations or Gates)

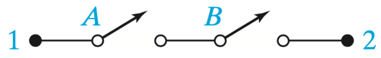


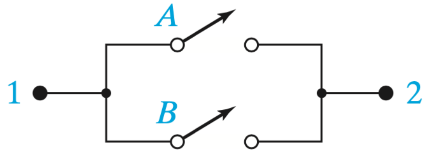
| a | y |
|---|---|
| 0 | 1 |
| 1 | 0 |

| a b | y |
|---|---|
| 0 0 | 0 |
| 0 1 | 0 |
| 1 0 | 0 |
| 1 1 | 1 |

| a b | y |
|---|---|
| 0 0 | 0 |
| 0 1 | 1 |
| 1 0 | 1 |
| 1 1 | 1 |

These tables we are using to show the behavior of the logic functions are called *TRUTH TABLES*





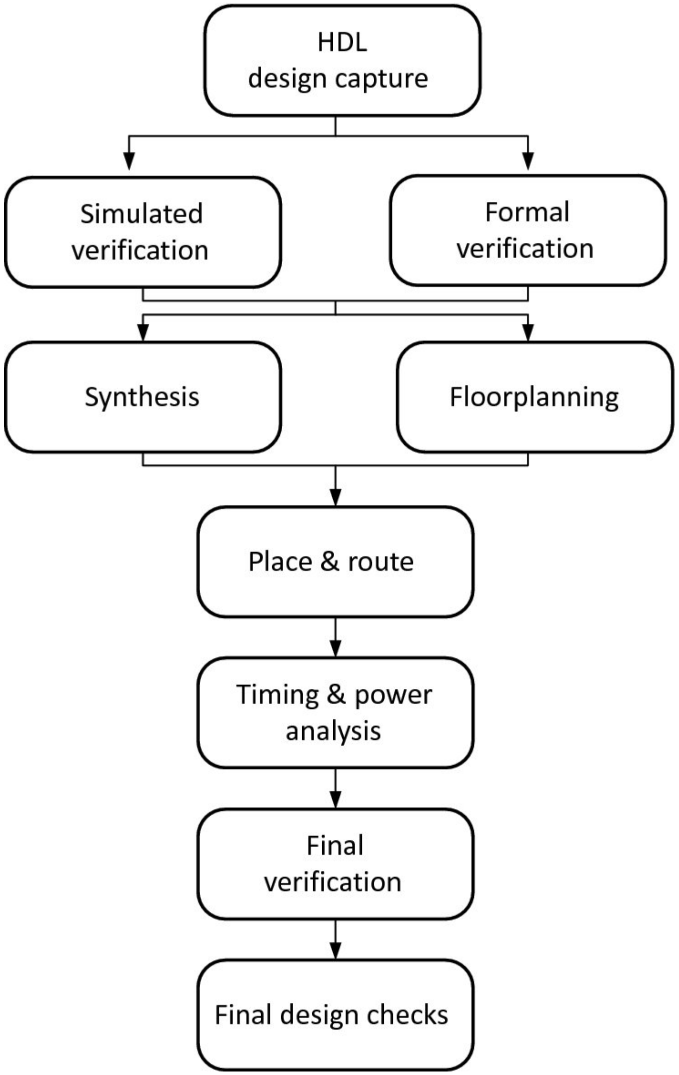
# Review

*Precedence of basic operations:*

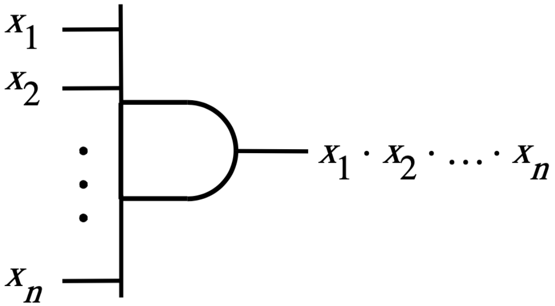
- Parenthesis
- NOT
- AND
- OR

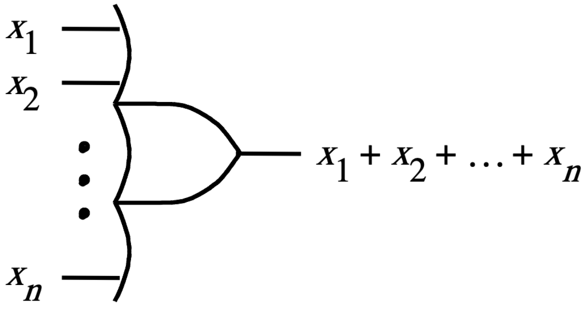
# Review

A typical CAD Flow

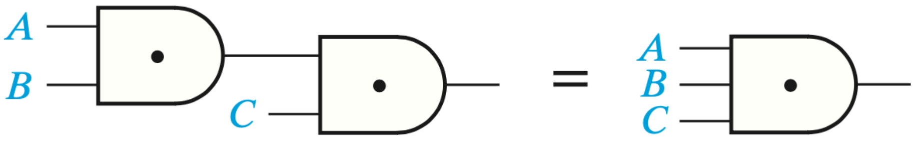


# *AND* and *OR* gates can have more than two inputs





*Example*



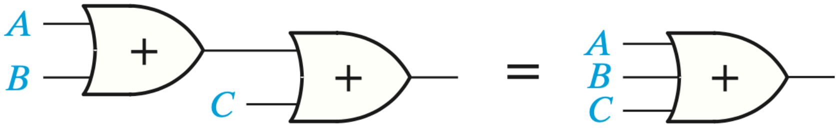
Think about the switch structure

| A | B | C | Y=A·B·C |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

*Example*

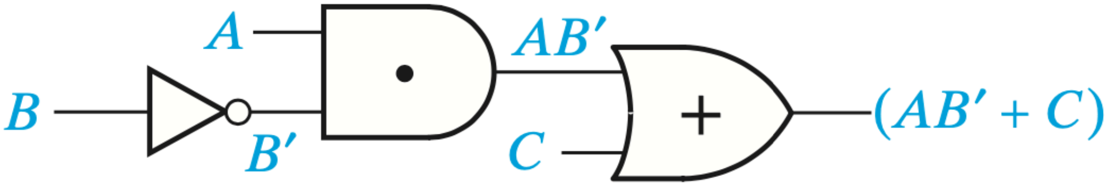
| A | B | C | Y=A+B+C |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

Think about switch structure



# “Composite” Boolean Functions (or Expressions)

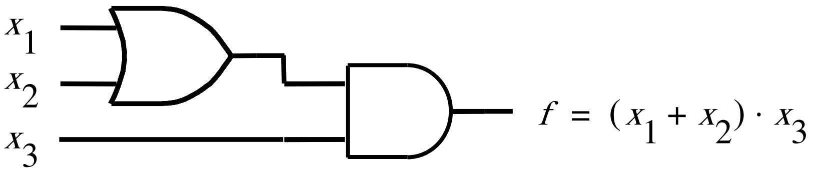
- All Boolean Functions are formed by applying the basic operations to one or more variables or constants



| A | B | C | Y=A·B’+C |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

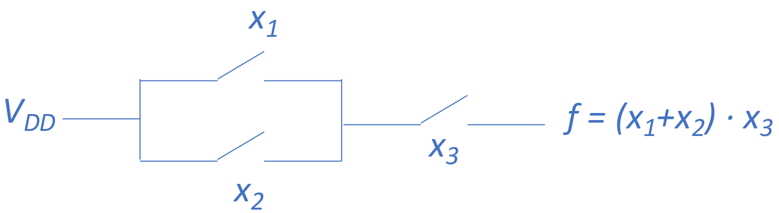
# “Composite” Logic Functions: another example

- So far we have seen 4 ways to express Logic Functions:
  - Switches
  - Equations (= expressions)
  - Truth Tables
    in general for N input variables the truth table has $2^N$ rows (there are $2^N$ combinations)



| $x_3\ x_2\ x_1$ | f |
|---|---|
| 0 0 0 | 0 |
| 0 0 1 | 0 |
| 0 1 0 | 0 |
| 0 1 1 | 0 |
| 1 0 0 | 0 |
| 1 0 1 | 1 |
| 1 1 0 | 1 |
| 1 1 1 | 1 |

  - Gates



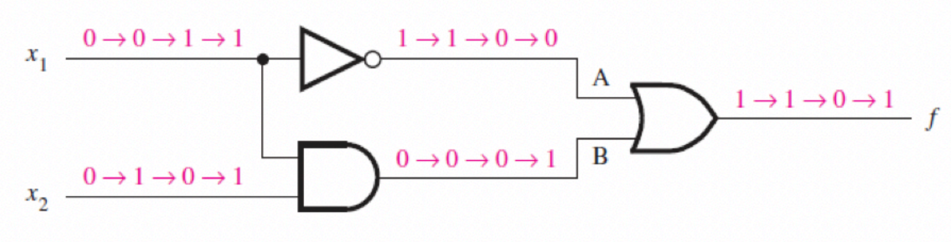
  - There is a fifth way called Karnaugh-maps (coming soon!)

# Analysis and Synthesis of Logic Functions

***Analysis***
Given a logic function, determine its behavior

***Synthesis***
Given a desired behavior, design the logic function that implements it

# Analysis of Logic Functions (1)



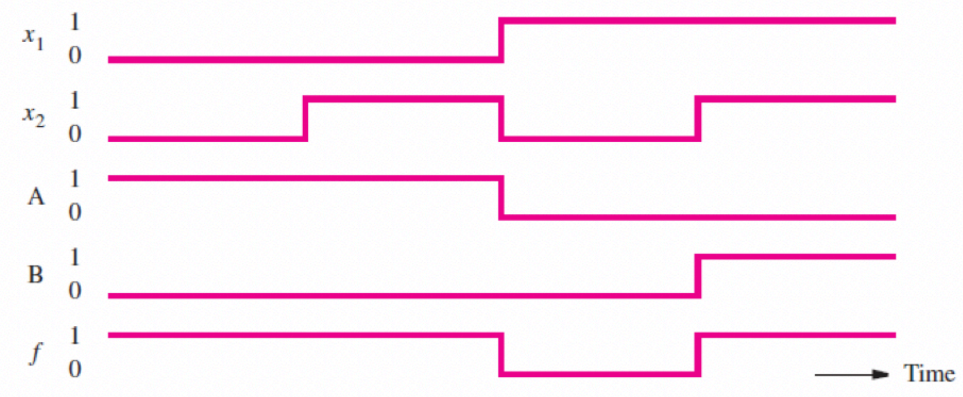
(a) Network that implements $f = \bar{x}_1 + x_1 \cdot x_2$

$$A = \overline{x_1} \quad B = x_1 \cdot x_2$$

| $x_1$ $x_2$ | $f = A + B$ |
|---|---|
| 0 0 | *1* |
| 0 1 | *1* |
| 1 0 | *0* |
| 1 1 | *1* |

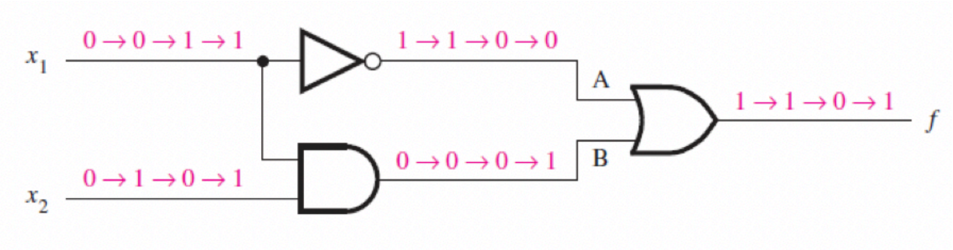
| A | B |
|---|---|
| 1 | 0 |
| 1 | 0 |
| 0 | 0 |
| 0 | 1 |

(b) Truth Table



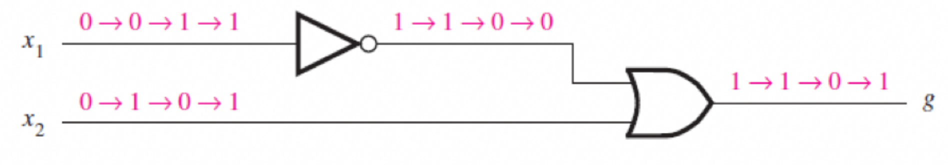
(c) Timing diagram

# Analysis of Logic Functions (2)


$x_1$
$0 \rightarrow 0 \rightarrow 1 \rightarrow 1$
$1 \rightarrow 1 \rightarrow 0 \rightarrow 0$
A
$x_2$
$0 \rightarrow 1 \rightarrow 0 \rightarrow 1$
$0 \rightarrow 0 \rightarrow 0 \rightarrow 1$
B
$1 \rightarrow 1 \rightarrow 0 \rightarrow 1$
$f$

(a) Network that implements $f = \bar{x}_1 + x_1 \cdot x_2$

*f and g behave exactly the same !*
*f and g are equivalent, but …*
*g is less "expensive" than f*

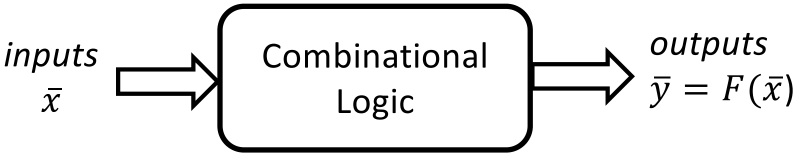

$x_1$
$0 \rightarrow 0 \rightarrow 1 \rightarrow 1$
$1 \rightarrow 1 \rightarrow 0 \rightarrow 0$
$x_2$
$0 \rightarrow 1 \rightarrow 0 \rightarrow 1$
$1 \rightarrow 1 \rightarrow 0 \rightarrow 1$
$g$

(d) Network that implements $g = \bar{x}_1 + x_2$

It would be nice to have some systematic method to find out the less "expensive" implementation of a given logic function !

# Types of Logic Circuits

- Combinational Circuits: the output values depend only on the present value of the inputs and not on past values.

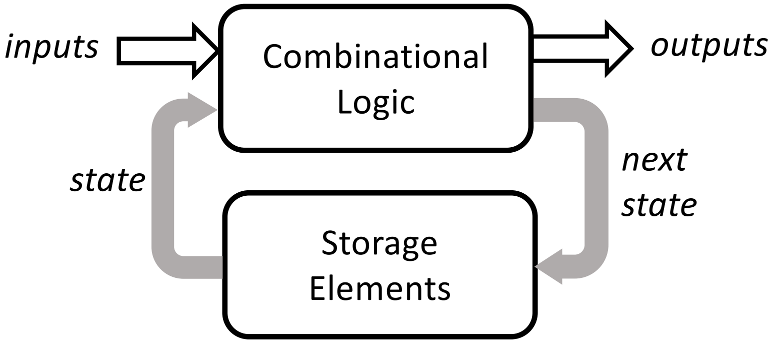


*inputs* $\bar{x}$

Combinational Logic

*outputs* $\bar{y} = F(\bar{x})$

- Sequential Circuits: the outputs depend on both the present and past input values



*inputs*

Combinational Logic

*outputs*

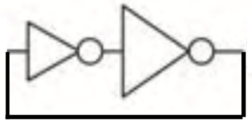
*state*

*next state*

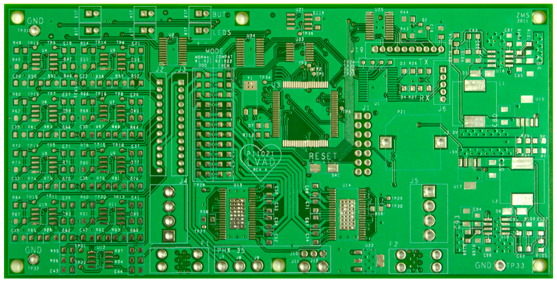
Storage Elements

As a consequence, combinational circuits are "memory less", while sequential circuits requires storage elements (latches or flip-flops).

To build storage elements we need a "feedback loop"

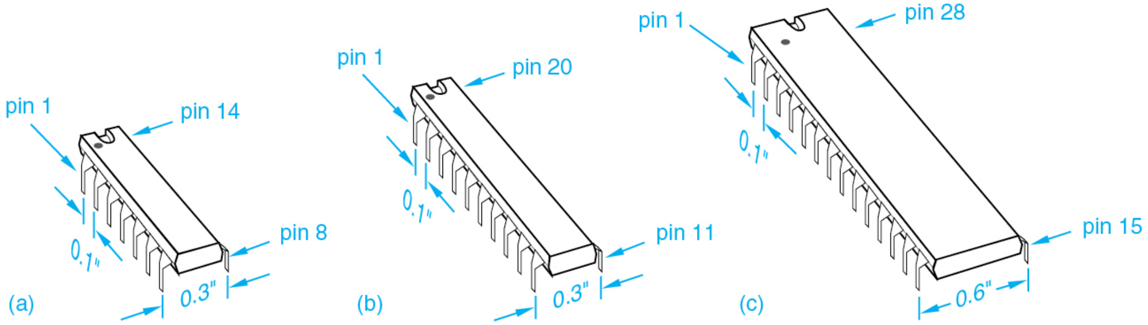


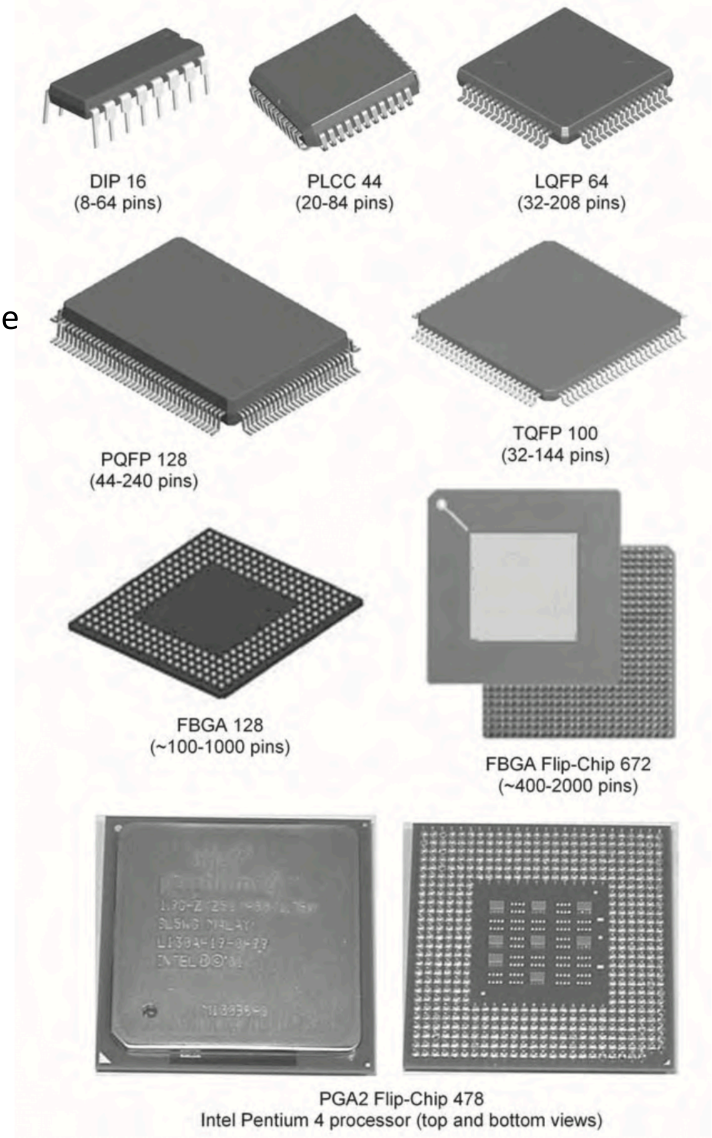
# PCBs and Integrated Circuits



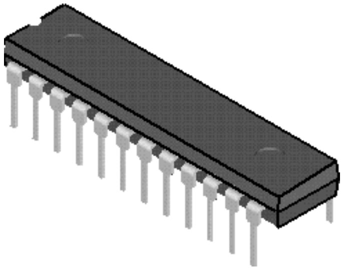
- DIP = dual Inline Package
- SMD = Surface Mount Devices
- PLCC = Plastic Leaded Chip Carrier
- LQFP = Low-Profile Quad Flat Package
- PQFP = Plastic Quad Flat Package
- TQFP = Thin Quad Flat package
- FBGA = Fine-Pitch Ball Grid Array
- PGA = Pin Grid Array

- Standard ICs
- Programmable ICs
- Application Specific ICs

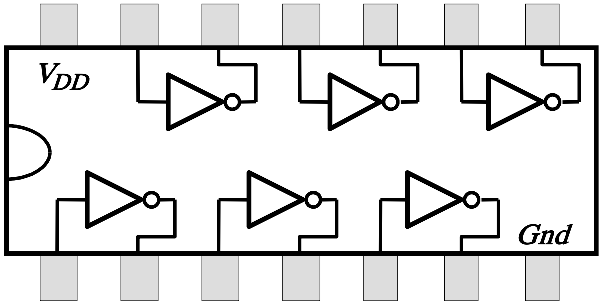




# Standard Chips (e.g. 7400-series)



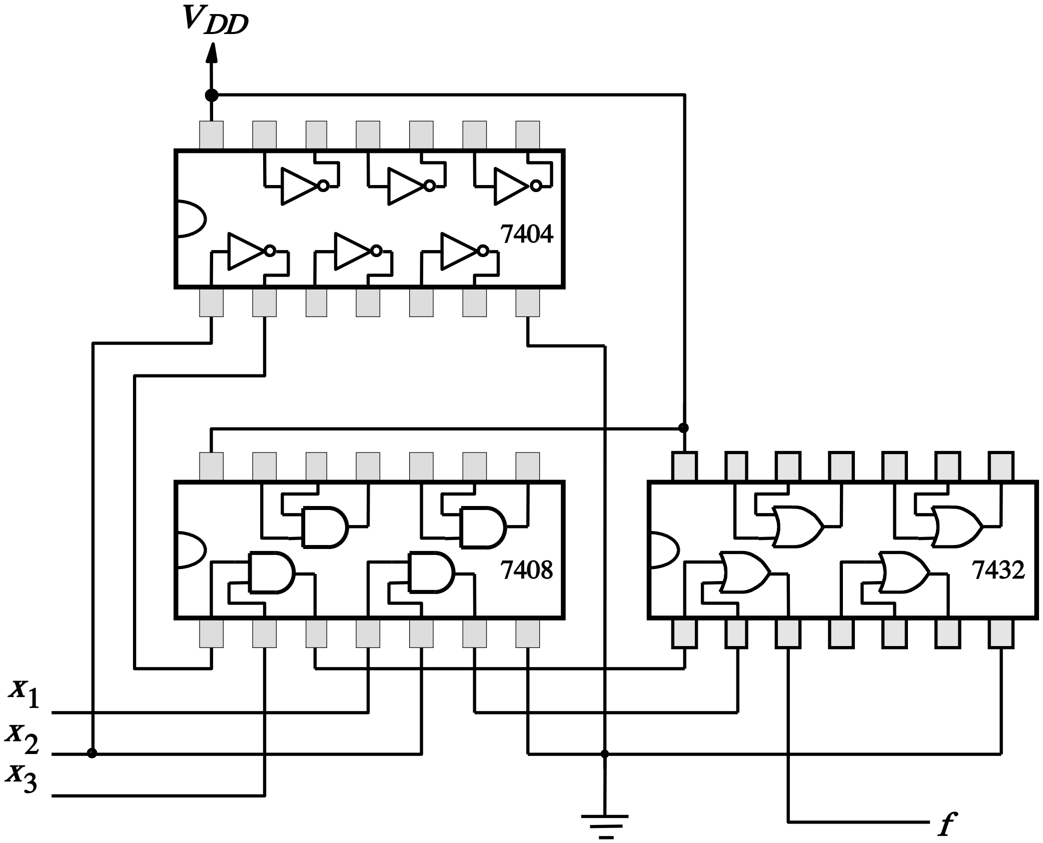
(a) Dual-inline package



(b) Structure of 7404 chip

# Implementation Example using Standard Chips

An implementation of $f = x_1x_2 + \overline{x}_2x_3$

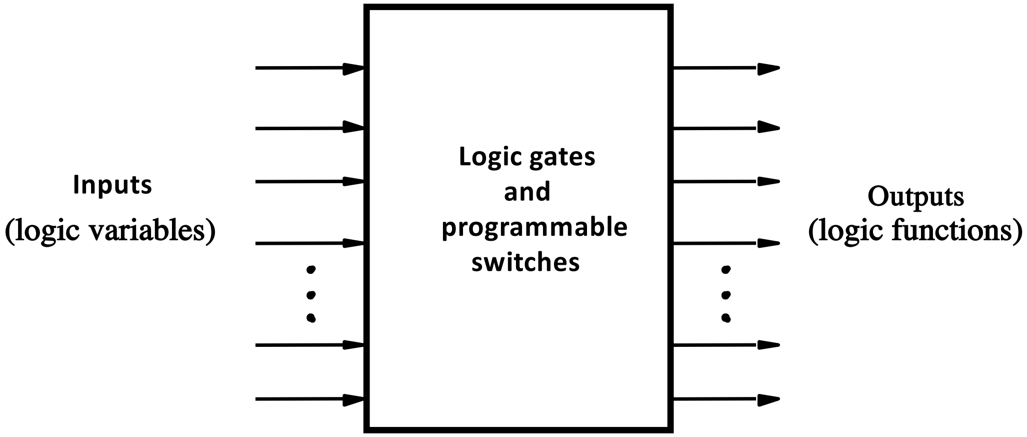


Limitations of 7400 series standard ships:

1. The function provided by each chip is fixed
2. Each chip only contains a few logic gates

# Programmable Logic Devices

- Programmable Logic Devices – chips that contain relatively large amounts of logic circuits with a structure that is not fixed (the structure can be customized)



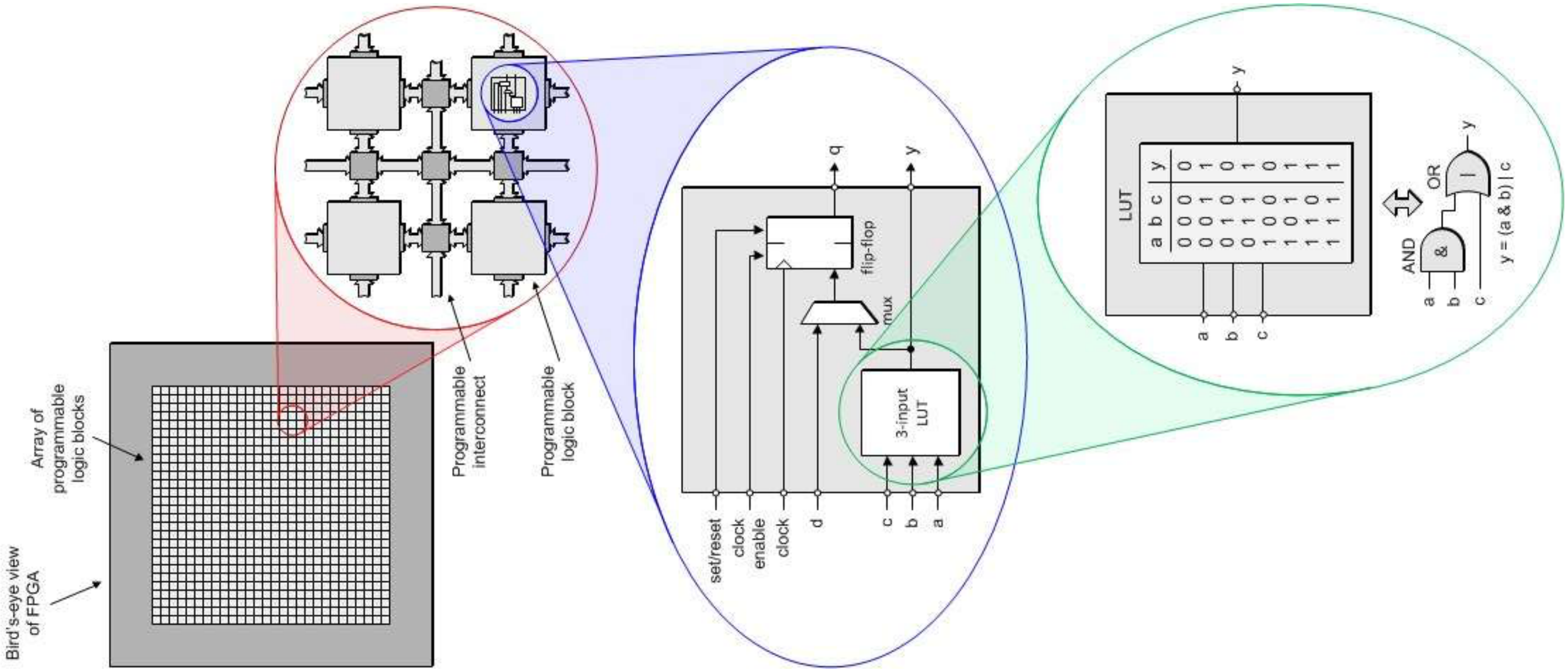
Programmable logic device as a black box

# Types of Programmable Logic Devices

- There are several types of Programmable Logic Devices commercially available:
  - Programmable Logic Array (PLA)
  - Programmable Array Logic (PAL)
  - Complex Programmable Logic Devices (CPLDs)
  - Field-Programmable Gate Arrays (FPGA)
- Logic circuit elements in programmable logic devices can be customized (that is programmed) through the use of CAD tools

# FPGA’s internal fabric

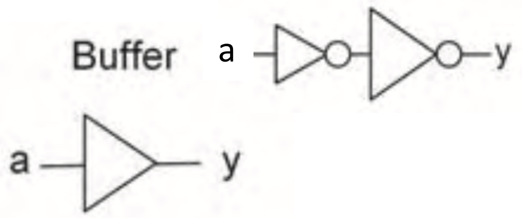
- Both the basic logic cells (LUT) an the interconnections are programmable



| a | b | c | y |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

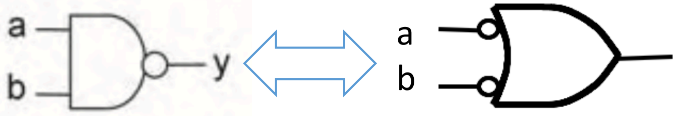
# Manipulating the Basic Boolean Functions

Buffer



| a | y |
|---|---|
| 0 | 0 |
| 1 | 1 |

NAND

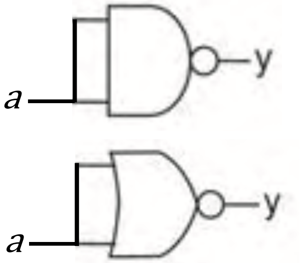


| a b | y |
|---|---|
| 0 0 | 1 |
| 0 1 | 1 |
| 1 0 | 1 |
| 1 1 | 0 |

| a b | a’ b’ | a’ + b’ |
|---|---|---|
| 0 0 | 1 1 | 1 |
| 0 1 | 1 0 | 1 |
| 1 0 | 0 1 | 1 |
| 1 1 | 0 0 | 0 |

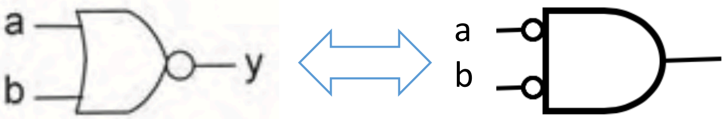
***if we want we can use NAND gates for everything !***

**INVERTER**



| a | y |
|---|---|
| 0 | 1 |
| 1 | 0 |

NOR



| a b | y |
|---|---|
| 0 0 | 1 |
| 0 1 | 0 |
| 1 0 | 0 |
| 1 1 | 0 |

| a b | a’ b’ | a’ * b’ |
|---|---|---|
| 0 0 | 1 1 | 1 |
| 0 1 | 1 0 | 0 |
| 1 0 | 0 1 | 0 |
| 1 1 | 0 0 | 0 |

***if we want we can use NOR gates for everything !***

# Boolean Algebra

***Objective:***
Systematic way of manipulating Boolean expression

- One-Variable Theorems

| | |
|---|---|
| $x \cdot 1 = x$ | $x + 0 = x$ |
| $x \cdot 0 = 0$ | $x + 1 = 1$ |
| $x \cdot x = x$ | $x + x = x$ |
| $x \cdot x' = 0$ | $x + x' = 1$ |
| $(x')' = x$ | |

# Boolean Algebra

- Two and Three-Variable Theorems

| | | |
|---|---|---|
| $x + x \cdot y = x$ | $x \cdot (x + y) = x$ | absorption (a.k.a. covering) |
| $x \cdot y + x \cdot y' = x$ | $(x + y)(x + y') = x$ | combining |
| $(x \cdot y)' = x' + y'$ | $(x + y)' = x' \cdot y'$ | De Morgan |
| $x \cdot y = y \cdot x$ | $x + y = y + x$ | commutative |
| $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ | $(x + y) + z = x + (y + z)$ | associative |
| $x \cdot (y + z) = x \cdot y + x \cdot z$ | $x + (y \cdot z) = (x + y) \cdot (x + z)$ | distributive |
| $x' \cdot y + x \cdot z = x' \cdot y + x \cdot z + y \cdot z$ | $(x' + y)(x + z) = (x' + y)(x + z)\,(y + z)$ | consensus (muxing) |

- Duality Principle

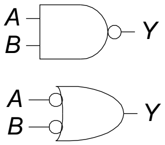
If a Boolean function *f* is true then the dual function $f_D$ is also true.
The dual of a logic function *f* is the function $f_D$ derived from f by swapping “+” with “·”, “·” with “+”, “0” with “1” and “1” with “0”.

# Boolean Algebra

- N-Variable Theorems

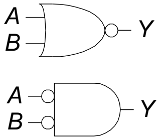
| | |
|---|---|
| $(x_1 + x_2 + x_3 + \cdots + x_N)' = x_1' \cdot x_2' \cdot x_3' \cdot \cdots \cdot x_N'$ | De Morgan Theorem |
| $(x_1 \cdot x_2 \cdot x_3 \cdot \cdots \cdot x_N)' = x_1' + x_2' + x_3' + \cdots + x_N'$ | |
| $f(x_1, x_2, x_3, \dots, x_N) = x_1' \cdot f(0, x_2, x_3, \dots, x_N) + x_1 \cdot f(1, x_2, x_3, \dots, x_N)$ | Shannon Expansion Theorem |
| $f(x_1, x_2, x_3, \dots, x_N) = [x_1' + f(1, x_2, x_3, \dots, x_N)] \cdot [x_1 + f(0, x_2, x_3, \dots, x_N)]$ | |

DeMorgan Theorem:
The **complement** of the **product** is the **sum** of the **complements.**



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

Dual:
The **complement** of the **sum** is the **product** of the **complements**.

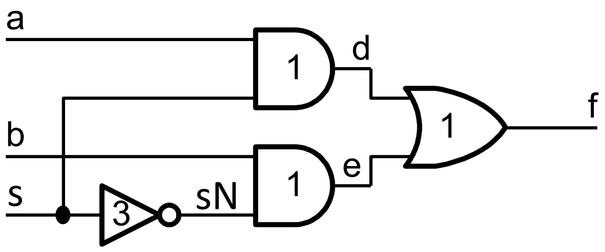


$$Y = \overline{A + B} = \bar{A} \cdot \bar{B}$$

# Next Time

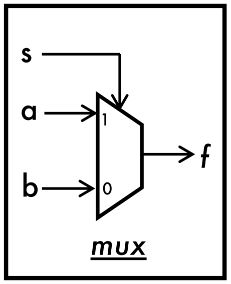
- Standard Form Representations of Logic Functions:
    - SOP
    - POS
    - min terms
    - max terms

# Consensus (a practical perspective)

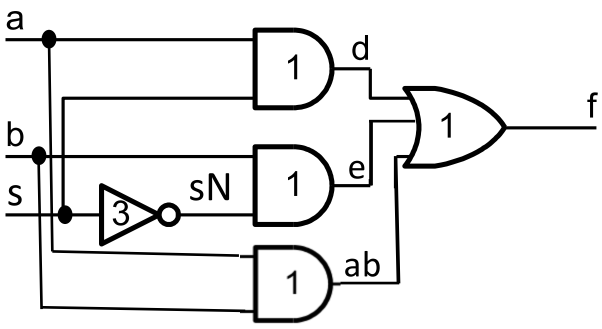


$f = a \cdot s + b \cdot \bar{s}$

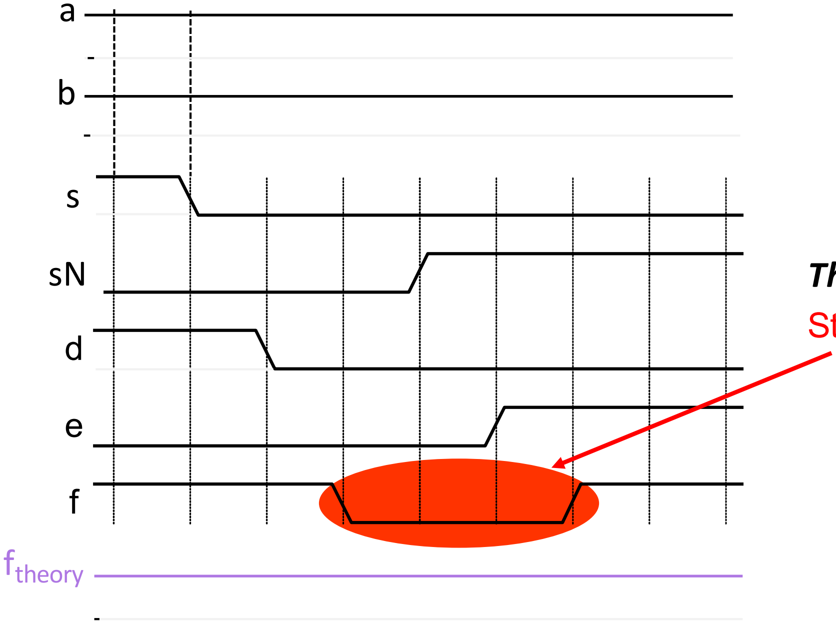
⇔



| s b a | f |
|---|---|
| 0 0 0 | 0 |
| 0 0 1 | 0 |
| 0 1 0 | 1 |
| 0 1 1 | 1 |
| 1 0 0 | 0 |
| 1 0 1 | 1 |
| 1 1 0 | 0 |
| 1 1 1 | 1 |

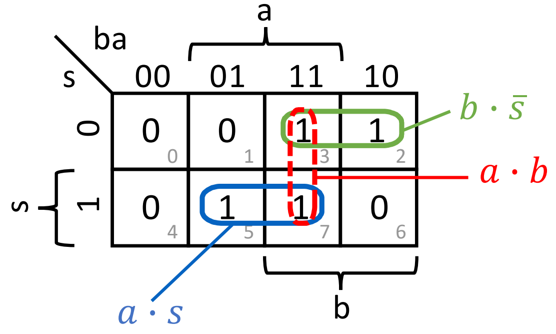


$f = a \cdot s + b \cdot \bar{s} + a \cdot b$



***There is a "Glitch"***

Static-1 hazard



| s \ ba | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 (0) | 0 (1) | 1 (3) | 1 (2) |
| 1 | 0 (4) | 1 (5) | 1 (7) | 0 (6) |

*Example.* *Prove Combining Theorem*

$$x \cdot y + x \cdot y' = x \cdot (y + y') = x \cdot 1 = x$$
$$(x + y) \cdot (x + y') = x \cdot x + x \cdot y + x \cdot y' + y \cdot y' = x + x \cdot (y + y') + 0 = x + x \cdot 1 + 0 = x$$

*Example.* *Prove Absorption Theorem*

$$x + x \cdot y = x \cdot (y + y') + x \cdot y = x \cdot y + x \cdot y' + x \cdot y = x \cdot y + x \cdot y' = x$$
$$x \cdot (x + y) = x \cdot x + x \cdot y = x + x \cdot y = x \cdot (y + y') + x \cdot y = x \cdot y + x \cdot y' + x \cdot y = x \cdot y + x \cdot y' = x$$

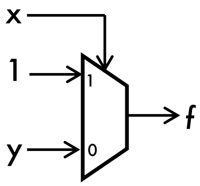
*Example.* *Prove Following Property*

*P1:* $x + \bar{x} \cdot y = x + y$ <-- Very easy to see from truth table or if we notice that it is a mux between 1 and y with x as selection (not so easy using equations)

*P1.D:* $x \cdot (\bar{x} + y) = x \cdot y$

| | x y | f |
|---|---|---|
| | 0 0 | 0 |
| $x'y$ | 0 1 | 1 |
| $x$ | 1 0 | 1 |
| | 1 1 | 1 |

*OR*



*Since P1 is not so easy to prove it may be a property worth remembering !*
*Let's call it "Simplification" Theorem.*

*= complement of missing term(s)*

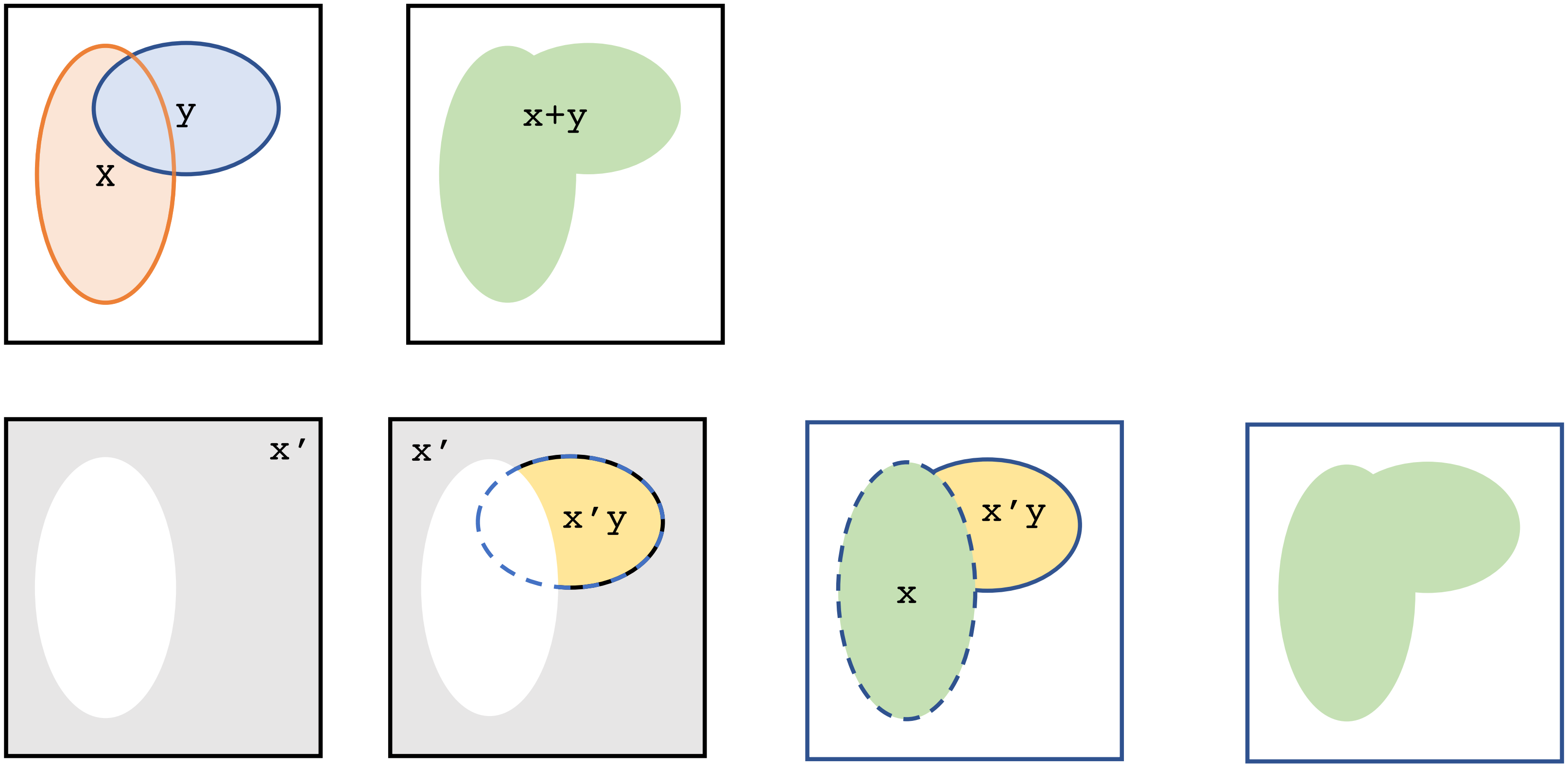
In other words remember that $f = \bar{\bar{f}}$

*P1 -->* $x + x'y = x(y + y') + x'y = xy + xy' + x'y = (x'y')' = (x + y)'' = x + y$

*P1.D -->* $x \cdot (x' + y) = x \cdot x' + x \cdot y = 0 + x \cdot y = x \cdot y$

*De Morgan*

*An addition tool to gain insight into a logic equation is to use Venn Diagrams from set theory*



x + x'y = x + y

# How to Prove ?

- **Method 1:** Perfect induction
- **Method 2:** Use other theorems and axioms to simplify the equation
  - Make one side of the equation look like the other

# What does Perfect Induction mean ?

- Despite the “fancy” name this is the most obvious solution
- Also called: **proof by exhaustion**
- Check every possible input value
- If two expressions produce the same value for every possible input combination, the expressions are equal

# Standard Forms

CPEN 230 – Introduction to Digital Logic

# Review: Boolean Algebra

**One-Variable Theorems**

| | |
|---|---|
| $x \cdot 1 = x$ | $x + 0 = x$ |
| $x \cdot 0 = 0$ | $x + 1 = 1$ |
| $x \cdot x = x$ | $x + x = x$ |
| $x \cdot x' = 0$ | $x + x' = 1$ |
| $(x')' = x$ | |

**Two and Three- Variable Theorems**

| | | |
|---|---|---|
| $x + x \cdot y = x$ | $x \cdot (x + y) = x$ | absorption (a.k.a. covering) |
| $x \cdot y + x \cdot y' = x$ | $(x + y)(x + y') = x$ | combining |
| $x + x' \cdot y = x + y$ | $x \cdot (x' + y) = x \cdot y$ | “simplification” |
| $(x \cdot y)' = x' + y'$ | $(x + y)' = x' \cdot y'$ | De Morgan |
| $x \cdot y = y \cdot x$ | $x + y = y + x$ | commutative |
| $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ | $(x + y) + z = x + (y + z)$ | associative |
| $x \cdot (y + z) = x \cdot y + x \cdot z$ | $x + (y \cdot z) = (x + y) \cdot (x + z)$ | distributive |
| $x' \cdot y + x \cdot z = x' \cdot y + x \cdot z + y \cdot z$ | $(x' + y)(x + z) = (x' + y)(x + z)\ (y + z)$ | consensus (muxing) |

# Review: Boolean Algebra

**Duality Principle**

If a Boolean function $f$ is true then the dual function $f_D$ is also true. The dual of a logic function $f$ is the function $f_D$ derived from f by swapping "+" with "$\cdot$", "$\cdot$" with "+", "0" with "1" and "1" with "0".

**N-Variable Theorems**

| | |
|---|---|
| $(x_1 + x_2 + x_3 + \cdots + x_N)' = x_1{}' \cdot x_2{}' \cdot x_3{}' \cdot \cdots \cdot x_N{}'$ | De Morgan Theorem |
| $(x_1 \cdot x_2 \cdot x_3 \cdot \cdots \cdot x_N)' = x_1{}' + x_2{}' + x_3{}' + \cdots + x_N{}'$ | |
| $f(x_1, x_2, x_3, \dots, x_N) = x_1{}' \cdot f(0, x_2, x_3, \dots, x_N) + x_1 \cdot f(1, x_2, x_3, \dots, x_N)$ | Shannon Expansion Theorem |
| $f(x_1, x_2, x_3, \dots, x_N) = [x_1{}' + f(1, x_2, x_3, \dots, x_N)] \cdot [x_1 + f(0, x_2, x_3, \dots, x_N)]$ | |

# Truth Tables: min-terms and max-terms

*For an N-input function a truth table has $2^N$ rows (= $2^N$ minterms = $2^N$ maxterms)*

| Row # | $x_2\ x_1\ x_0$ | f | Minterm | Maxterm |
|---|---|---|---|---|
| 0 | 000 | | $m_0 = \overline{x_2} \cdot \overline{x_1} \cdot \overline{x_0}$ | $M_0 = x_2 + x_1 + x_0 = \overline{m_0}$ |
| 1 | 001 | | $m_1 = \overline{x_2} \cdot \overline{x_1} \cdot x_0$ | $M_1 = x_2 + x_1 + \overline{x_0} = \overline{m_1}$ |
| 2 | 010 | | $m_2 = \overline{x_2} \cdot x_1 \cdot \overline{x_0}$ | $M_2 = x_2 + \overline{x_1} + x_0 = \overline{m_2}$ |
| 3 | 011 | | $m_3 = \overline{x_2} \cdot x_1 \cdot x_0$ | $M_3 = x_2 + \overline{x_1} + \overline{x_0} = \overline{m_3}$ |
| 4 | 100 | | $m_4 = x_2 \cdot \overline{x_1} \cdot \overline{x_0}$ | $M_4 = \overline{x_2} + x_1 + x_0 = \overline{m_4}$ |
| 5 | 101 | | $m_5 = x_2 \cdot \overline{x_1} \cdot x_0$ | $M_5 = \overline{x_2} + x_1 + \overline{x_0} = \overline{m_5}$ |
| 6 | 110 | | $m_6 = x_2 \cdot x_1 \cdot \overline{x_0}$ | $M_6 = \overline{x_2} + \overline{x_1} + x_0 = \overline{m_6}$ |
| 7 | 111 | | $m_7 = x_2 \cdot x_1 \cdot x_0$ | $M_7 = \overline{x_2} + \overline{x_1} + \overline{x_0} = \overline{m_7}$ |

# Terminology

- **Literal**
  Any occurrence of a variable, either in its direct form (e.g. $a$) or in its complemented form (e.g. $\bar{a}$)

- **Product Term**
  A product (AND operation) of two or more literals
  (e.g. $ab, a\bar{a}, \bar{a}b, \bar{a}bc, \ldots$)

- **Sum Term**
  A sum (OR operation) of two or more literals
  (e.g. $\bar{a} + b, \bar{a} + b, a + b + c, c + c, \ldots$)

- **Minterm**
  Given a function of N-input variables, a minterm is a product term of N literals with one literal for each input variable.
  (Example: given a function of three input variables $\bar{a}bc$ is a min term, but $\bar{a}b$ or $\bar{a}ab$ are not)

- **Maxterm**
  Given a function of N-input variables, a maxterm is a sum term of N literals with one literal for each input variable.

# Minterms and maxterms: example

| Row # | $x_2\ x_1\ x_0$ | f | Minterm | Maxterm |
|---|---|---|---|---|
| 0 | 000 | 1 | $m_0 = \overline{x_2} \cdot \overline{x_1} \cdot \overline{x_0}$ | $M_0 = x_2 + x_1 + x_0 = \overline{m_0}$ |
| 1 | 001 | 1 | $m_1 = \overline{x_2} \cdot \overline{x_1} \cdot x_0$ | $M_1 = x_2 + x_1 + \overline{x_0} = \overline{m_1}$ |
| 2 | 010 | 0 | $m_2 = \overline{x_2} \cdot x_1 \cdot \overline{x_0}$ | $M_2 = x_2 + \overline{x_1} + x_0 = \overline{m_2}$ |
| 3 | 011 | 0 | $m_3 = \overline{x_2} \cdot x_1 \cdot x_0$ | $M_3 = x_2 + \overline{x_1} + \overline{x_0} = \overline{m_3}$ |
| 4 | 100 | 0 | $m_4 = x_2 \cdot \overline{x_1} \cdot \overline{x_0}$ | $M_4 = \overline{x_2} + x_1 + x_0 = \overline{m_4}$ |
| 5 | 101 | 0 | $m_5 = x_2 \cdot \overline{x_1} \cdot x_0$ | $M_5 = \overline{x_2} + x_1 + \overline{x_0} = \overline{m_5}$ |
| 6 | 110 | 0 | $m_6 = x_2 \cdot x_1 \cdot \overline{x_0}$ | $M_6 = \overline{x_2} + \overline{x_1} + x_0 = \overline{m_6}$ |
| 7 | 111 | 0 | $m_7 = x_2 \cdot x_1 \cdot x_0$ | $M_7 = \overline{x_2} + \overline{x_1} + \overline{x_0} = \overline{m_7}$ |

If a function *f* is specified in the form of a Truth Table, then an expression that realizes *f* can be obtained either by considering the rows in the table for which f=1 or by considering the rows for which *f*=0

$$
\begin{aligned}
f &= m_0 + m_1 = (m_2 + m_3 + m_4 + m_5 + m_6 + m_7)' = (\overline{m_2} \cdot \overline{m_3} \cdot \overline{m_4} \cdot \overline{m_5} \cdot \overline{m_6} \cdot \overline{m_7})'' \\
&= M_2 \cdot M_3 \cdot M_4 \cdot M_5 \cdot M_6 \cdot M_7
\end{aligned}
$$

# Minterms and maxterms: example

| Row # | $x_2$ $x_1$ $x_0$ | f | Minterm | Maxterm |
|---|---|---|---|---|
| 0 | 000 | 1 | $m_0 = \overline{x_2} \cdot \overline{x_1} \cdot \overline{x_0}$ | $M_0 = x_2 + x_1 + x_0 = \overline{m_0}$ |
| 1 | 001 | 1 | $m_1 = \overline{x_2} \cdot \overline{x_1} \cdot x_0$ | $M_1 = x_2 + x_1 + \overline{x_0} = \overline{m_1}$ |
| 2 | 010 | 0 | $m_2 = \overline{x_2} \cdot x_1 \cdot \overline{x_0}$ | $M_2 = x_2 + \overline{x_1} + x_0 = \overline{m_2}$ |
| 3 | 011 | 0 | $m_3 = \overline{x_2} \cdot x_1 \cdot x_0$ | $M_3 = x_2 + \overline{x_1} + \overline{x_0} = \overline{m_3}$ |
| 4 | 100 | 0 | $m_4 = x_2 \cdot \overline{x_1} \cdot \overline{x_0}$ | $M_4 = \overline{x_2} + x_1 + x_0 = \overline{m_4}$ |
| 5 | 101 | 0 | $m_5 = x_2 \cdot \overline{x_1} \cdot x_0$ | $M_5 = \overline{x_2} + x_1 + \overline{x_0} = \overline{m_5}$ |
| 6 | 110 | 0 | $m_6 = x_2 \cdot x_1 \cdot \overline{x_0}$ | $M_6 = \overline{x_2} + \overline{x_1} + x_0 = \overline{m_6}$ |
| 7 | 111 | 0 | $m_7 = x_2 \cdot x_1 \cdot x_0$ | $M_7 = \overline{x_2} + \overline{x_1} + \overline{x_0} = \overline{m_7}$ |

$$f = \sum m(0,1) = \prod M\,(2,3,4,5,6,7)$$

# Sum-of-Products (SOPs) and Product-of-Sums (POSs)

Any Boolean function *f* can be expressed either as a sum of product terms (SOPs) or as a product of sum terms (POSs)

- If all the product terms in the SOP are minterms then the expression is called a normal (or canonical or standard) form SOP
- If all the sum terms in the POS are maxterms then the expression is called a normal (or canonical or standard) form POS

combining theorem

*Example*

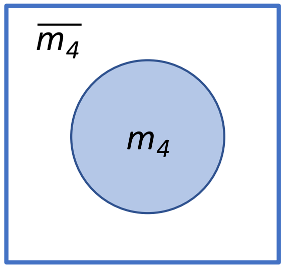
$$f(x_2, x_2, x_1) = \sum m(0,1) = x_2{}' x_1{}' x_0' + x_2{}' x_1' x_0 = (x_2{}' x_1')\overbrace{(x_0' + x_0)}^{=1} = x_2{}' x_1'$$

canonical SOP form

non-canonical SOP form

# Minterms and maxterms

| Row # | $x_2\,x_1\,x_0$ | f | Minterm | Maxterm |
|---|---|---|---|---|
| 0 | 000 | 0 | $m_0 = \overline{x_2} \cdot \overline{x_1} \cdot \overline{x_0}$ | $M_0 = x_2 + x_1 + x_0 = \overline{m_0}$ |
| 1 | 001 | 0 | $m_1 = \overline{x_2} \cdot \overline{x_1} \cdot x_0$ | $M_1 = x_2 + x_1 + \overline{x_0} = \overline{m_1}$ |
| 2 | 010 | 0 | $m_2 = \overline{x_2} \cdot x_1 \cdot \overline{x_0}$ | $M_2 = x_2 + \overline{x_1} + x_0 = \overline{m_2}$ |
| 3 | 011 | 0 | $m_3 = \overline{x_2} \cdot x_1 \cdot x_0$ | $M_3 = x_2 + \overline{x_1} + \overline{x_0} = \overline{m_3}$ |
| 4 | 100 | 1 | $m_4 = x_2 \cdot \overline{x_1} \cdot \overline{x_0}$ | $M_4 = \overline{x_2} + x_1 + x_0 = \overline{m_4}$ |
| 5 | 101 | 0 | $m_5 = x_2 \cdot \overline{x_1} \cdot x_0$ | $M_5 = \overline{x_2} + x_1 + \overline{x_0} = \overline{m_5}$ |
| 6 | 110 | 0 | $m_6 = x_2 \cdot x_1 \cdot \overline{x_0}$ | $M_6 = \overline{x_2} + \overline{x_1} + x_0 = \overline{m_6}$ |
| 7 | 111 | 0 | $m_7 = x_2 \cdot x_1 \cdot x_0$ | $M_7 = \overline{x_2} + \overline{x_1} + \overline{x_0} = \overline{m_7}$ |



Each minterm “touches” only one row of the truth table (e.g. $m_4$)

Each maxterm “touches” all but one row of the table (e.g. $M_4 = \overline{m_4}$ *so it touches:* $m_0$ ; $m_1$ ; $m_2$ ; $m_3$ ; $m_5$ ; $m_6$ *and* $m_7$)

# Simplyfing an Equation

Reducing an equation to the **fewest number of implicants**, where each implicant has the **fewest literals**

**Recall:**

– Implicant: product of literals

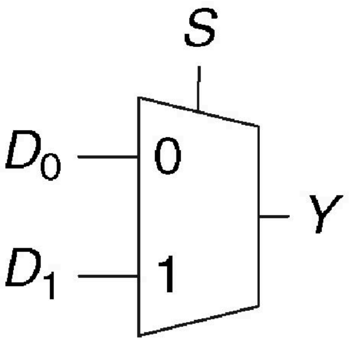
***ABC, AC, BC***

– Literal: variable or its complement

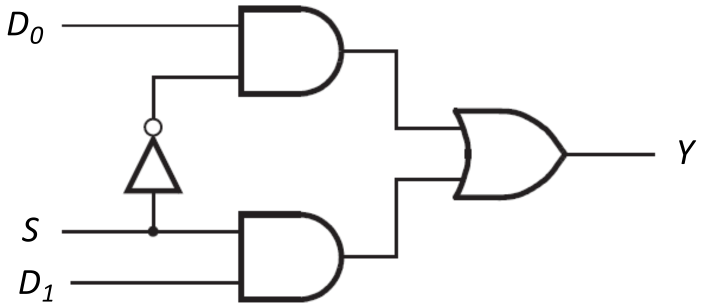
***A, A', B, B', C, C'***

*Also called **minimizing** the equation*

# Design Example: Multiplexer



```
if (S == 0)
  Y <= D0;
else
  Y <= D1;
```



| $S$ | $D_1$ | $D_0$ | $Y$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

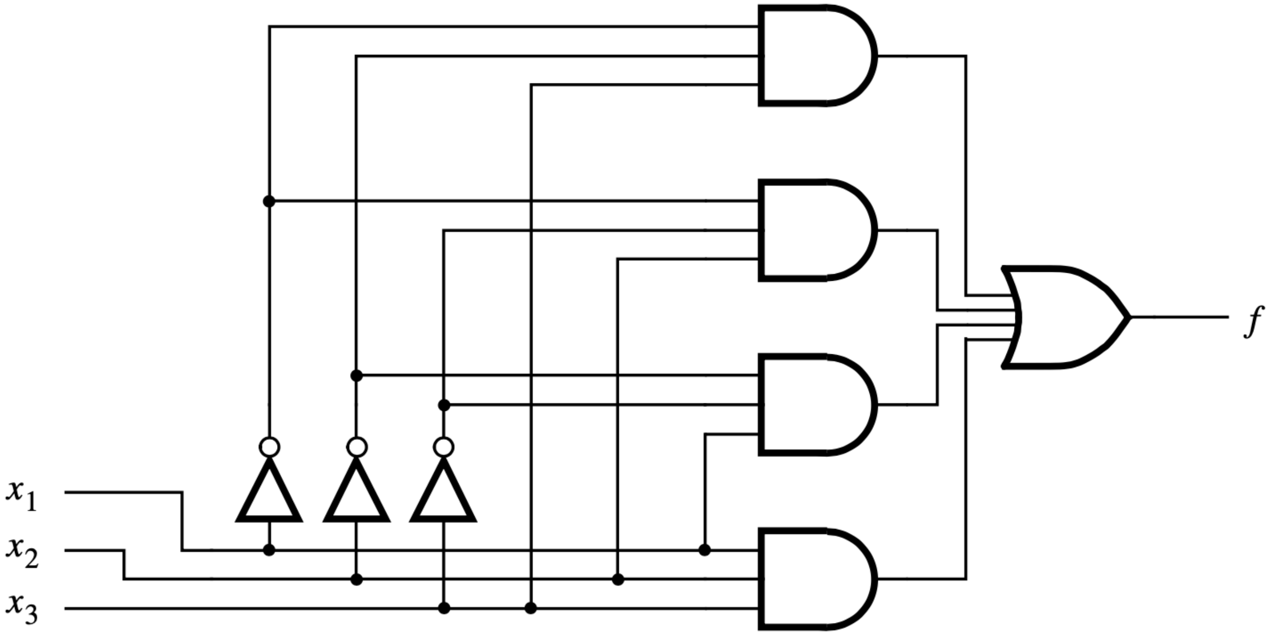
$$Y = \sum m(1,3,6,7) =$$

$$= \bar{S} \cdot \overline{D_1} \cdot D_0 + \bar{S} \cdot D_1 \cdot D_0 + S \cdot D_1 \cdot \overline{D_0} + S \cdot D_1 \cdot D_0 =$$

$$= \bar{S} \cdot D_0 \cdot \underbrace{(\overline{D_1} + D_1)}_{=1} + S \cdot D_1 \cdot \underbrace{(\overline{D_0} + D_0)}_{=1} = \bar{S} \cdot D_0 + S \cdot D_1$$

# Design Example: Three way light control

| $x_1$ | $x_2$ | $x_3$ | $f$ |
|---|---|---|---|
| 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 |

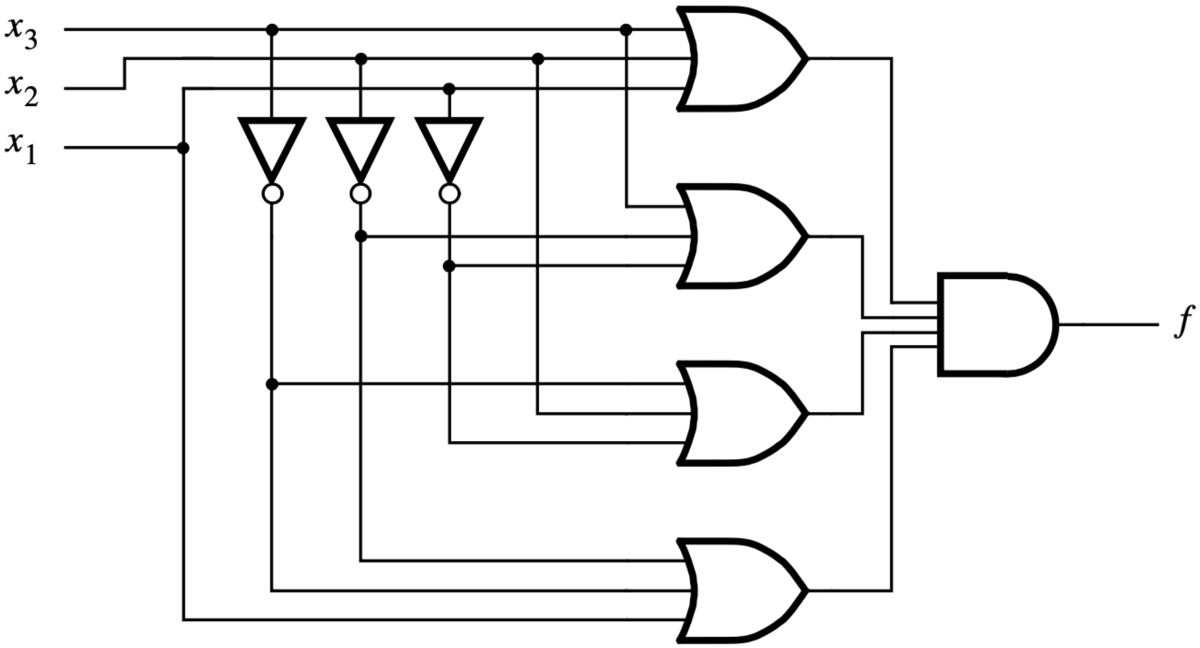


*SOP realization*

$$f(x_1, x_2, x_3) = m_1 + m_2 + m_4 + m_7 = \overline{x_1} \cdot \overline{x_2} \cdot x_3 + \overline{x_1} \cdot x_2 \cdot \overline{x_3} + x_1 \cdot \overline{x_2} \cdot \overline{x_3} + x_1 \cdot x_2 \cdot x_3$$

# Design Example: Three way light control

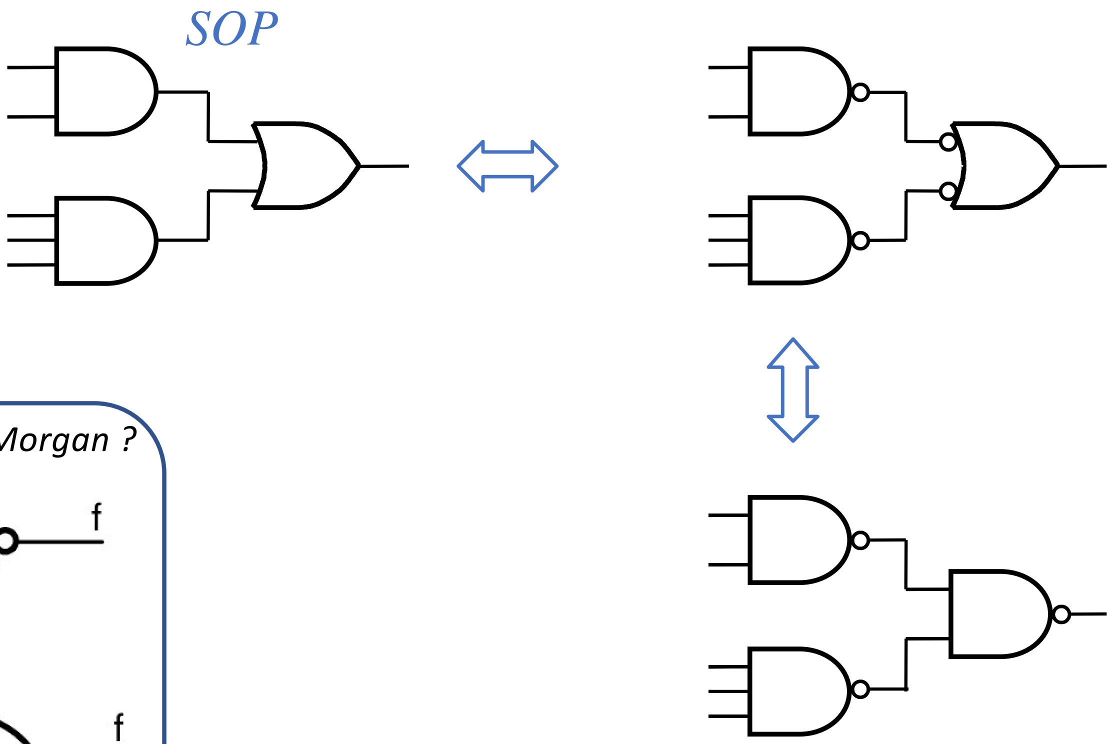
| $x_1$ | $x_2$ | $x_3$ | $f$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |



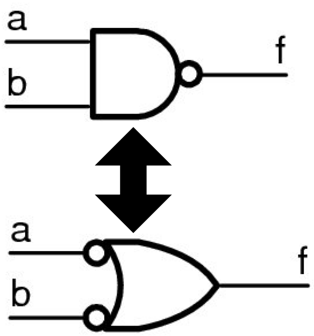
*POS realization*

$$f(x_1,x_2,x_3) = M_0 \cdot M_3 \cdot M_5 \cdot M_6 = (x_1+x_2+x_3)\cdot(x_1+\overline{x_2}+\overline{x_3})\cdot(\overline{x_1}+x_2+\overline{x_3})\cdot(\overline{x_1}+\overline{x_2}+x_3)$$

# NAND-NAND circuits (pushing bubbles)



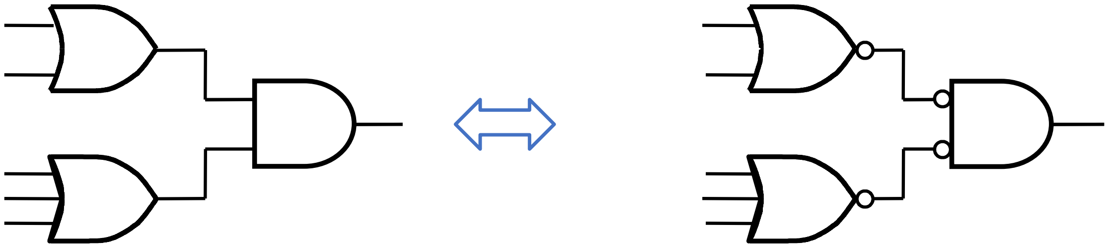
*Remember De Morgan ?*

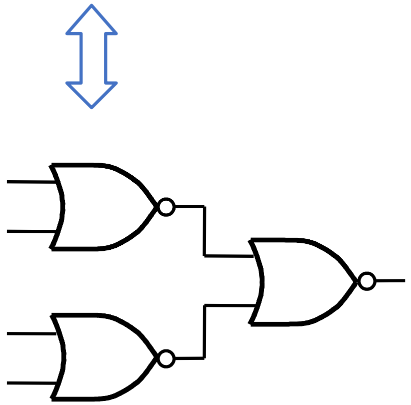


*Using NAND gates to implement a sum-of-products.*

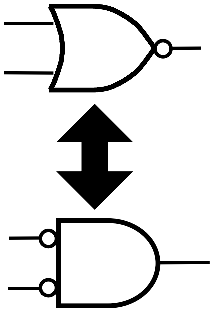
# NOR-NOR circuits (pushing bubbles)

*POS*





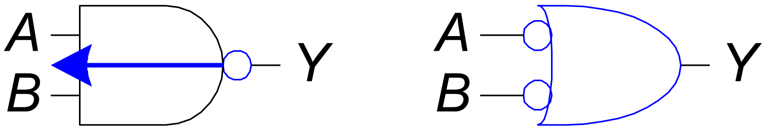
*Remember De Morgan ?*



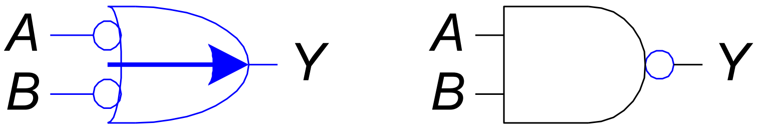
*Using NOR gates to implement a product-of sums.*

# Bubble Pushing

- **Backward:**
  - Body gets “squished” (changes shape)
  - Adds bubbles to inputs

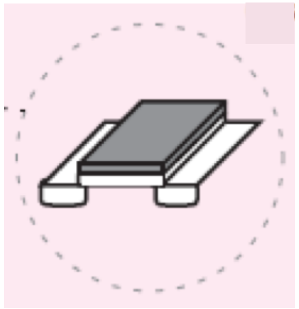


- **Forward:**
  - Body gets “squished” (changes shape)
  - Adds bubble to output

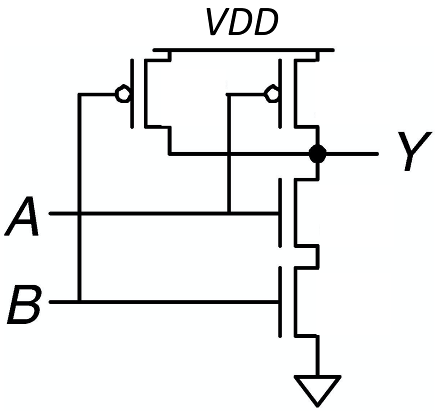


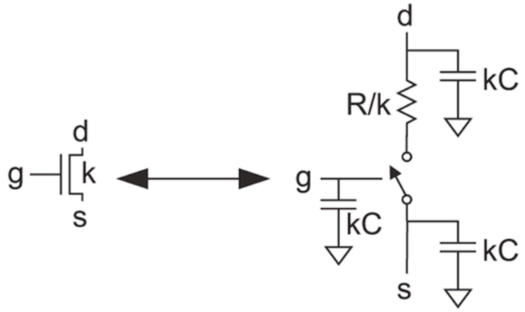
# NAND-NAND vs. NOR-NOR

- The propagation delay through a logic gate is proportional to the RC of the switches involved in delivering the desired logic level (either $V_{DD}$ or *GND*) to the output. The switches (MOS transistors) are not ideal: they have R and C associated to them (the PMOS transistors have more R than the NMOS transistors)

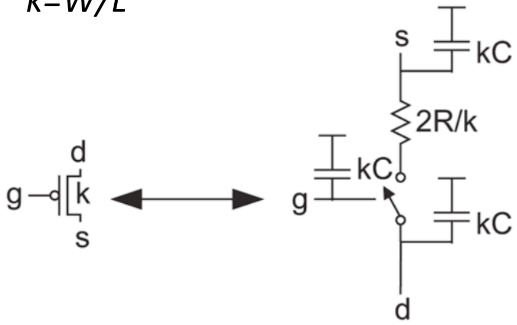


*2-input CMOS NAND gate*

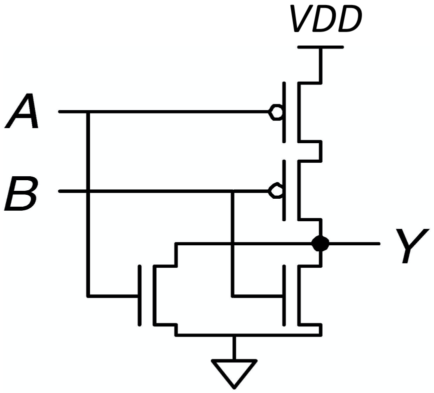




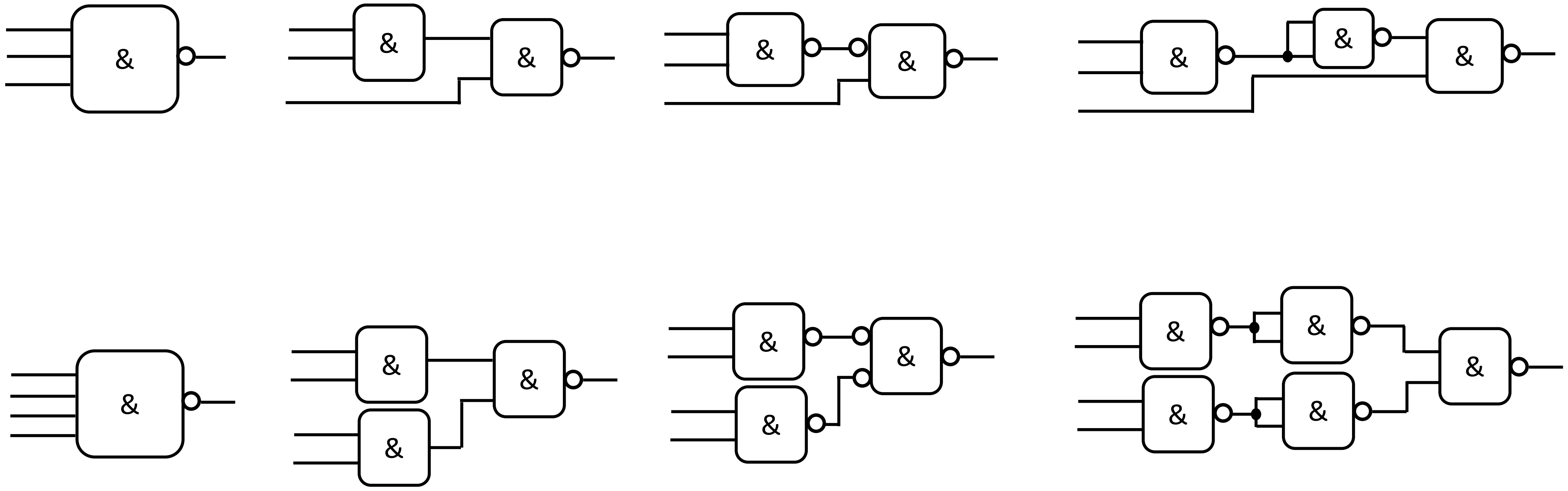
*k=W/L*



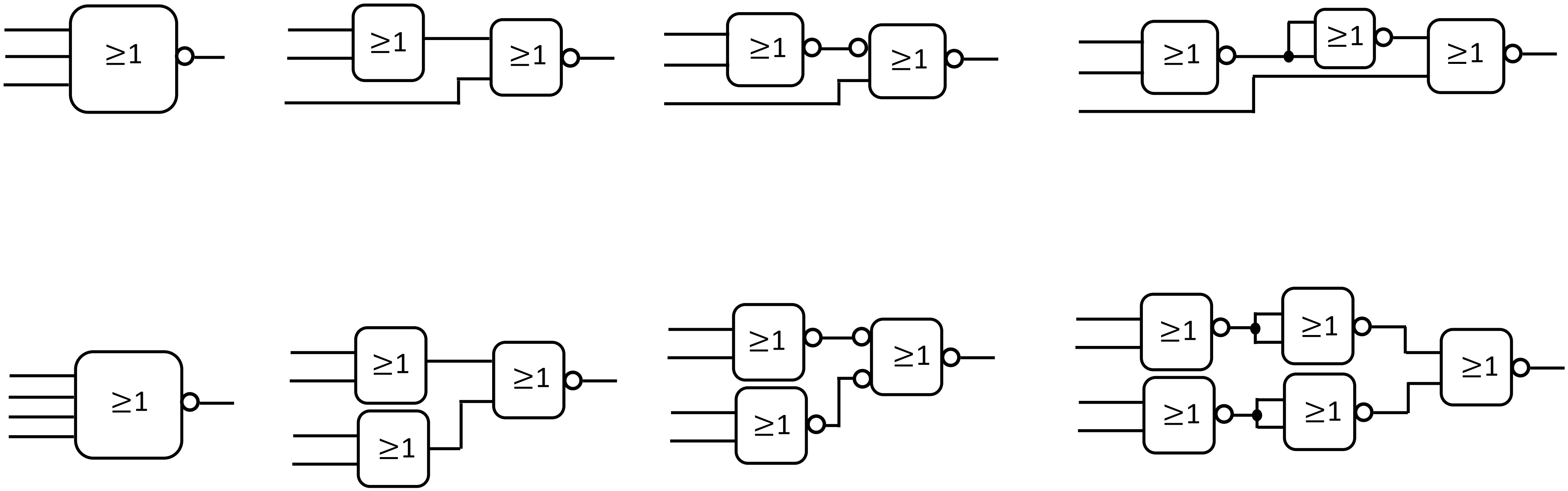
*2-input CMOS NOR gate*



# Building multiple-input NAND gates using 2-input NAND gates (pushing bubbles)



# Building multiple-input NOR gates using 2-input NOR gates (pushing bubbles)



# Next Time

- A first look at Verilog
- K-maps
- Minimization of Boolean Equations using K-maps
- Multiple Drivers (contention Value: X)
- Tri-state Buffer (high impedance state: Z)

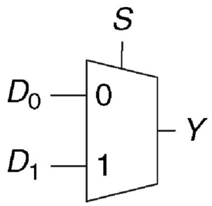
# K-maps and minimization

CPEN 230 – Introduction to Digital Logic

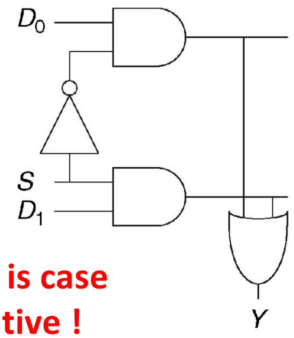
# Review: minterms, maxterms, SOP form and POS form

| Row # | $x_2$ $x_1$ $x_0$ | f | Minterm | Maxterm |
|---|---|---|---|---|
| 0 | 000 | 1 | $m_0 = \overline{x_2} \cdot \overline{x_1} \cdot \overline{x_0}$ | $M_0 = x_2 + x_1 + x_0 = \overline{m_0}$ |
| 1 | 001 | 1 | $m_1 = \overline{x_2} \cdot \overline{x_1} \cdot x_0$ | $M_1 = x_2 + x_1 + \overline{x_0} = \overline{m_1}$ |
| 2 | 010 | 0 | $m_2 = \overline{x_2} \cdot x_1 \cdot \overline{x_0}$ | $M_2 = x_2 + \overline{x_1} + x_0 = \overline{m_2}$ |
| 3 | 011 | 0 | $m_3 = \overline{x_2} \cdot x_1 \cdot x_0$ | $M_3 = x_2 + \overline{x_1} + \overline{x_0} = \overline{m_3}$ |
| 4 | 100 | 0 | $m_4 = x_2 \cdot \overline{x_1} \cdot \overline{x_0}$ | $M_4 = \overline{x_2} + x_1 + x_0 = \overline{m_4}$ |
| 5 | 101 | 0 | $m_5 = x_2 \cdot \overline{x_1} \cdot x_0$ | $M_5 = \overline{x_2} + x_1 + \overline{x_0} = \overline{m_5}$ |
| 6 | 110 | 0 | $m_6 = x_2 \cdot x_1 \cdot \overline{x_0}$ | $M_6 = \overline{x_2} + \overline{x_1} + x_0 = \overline{m_6}$ |
| 7 | 111 | 0 | $m_7 = x_2 \cdot x_1 \cdot x_0$ | $M_7 = \overline{x_2} + \overline{x_1} + \overline{x_0} = \overline{m_7}$ |

$$f = \sum m(0,1) = \prod M\ (2,3,4,5,6,7)$$



# A first look at Verilog



*Register Transfer Level (RTL) Verilog Code for synthesizing a 2:1 mux*

**Verilog is case sensitive !**

```
// filename: mux21.v
// two possible coding styles for implementing a 2:1 multiplexer

module mux21(s,d1,d0,f,y);
  input s, d0, d1;
  output f, y;

  reg y;

  assign f = (~s & d0) | (s & d1);

  always@ (s or d0 or d1)
    if (s == 0)
      y = d0;
    else
      y = d1;

endmodule
```

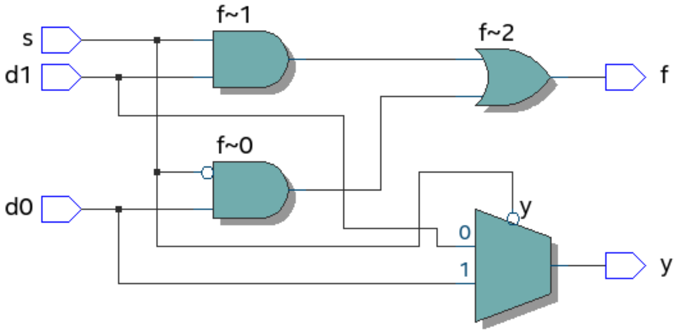
*blocking assignment*

*sensitivity list*

*concurrent coding style*

*always-based (procedural) coding style (higher level of abstraction)*

*blocking assignment*



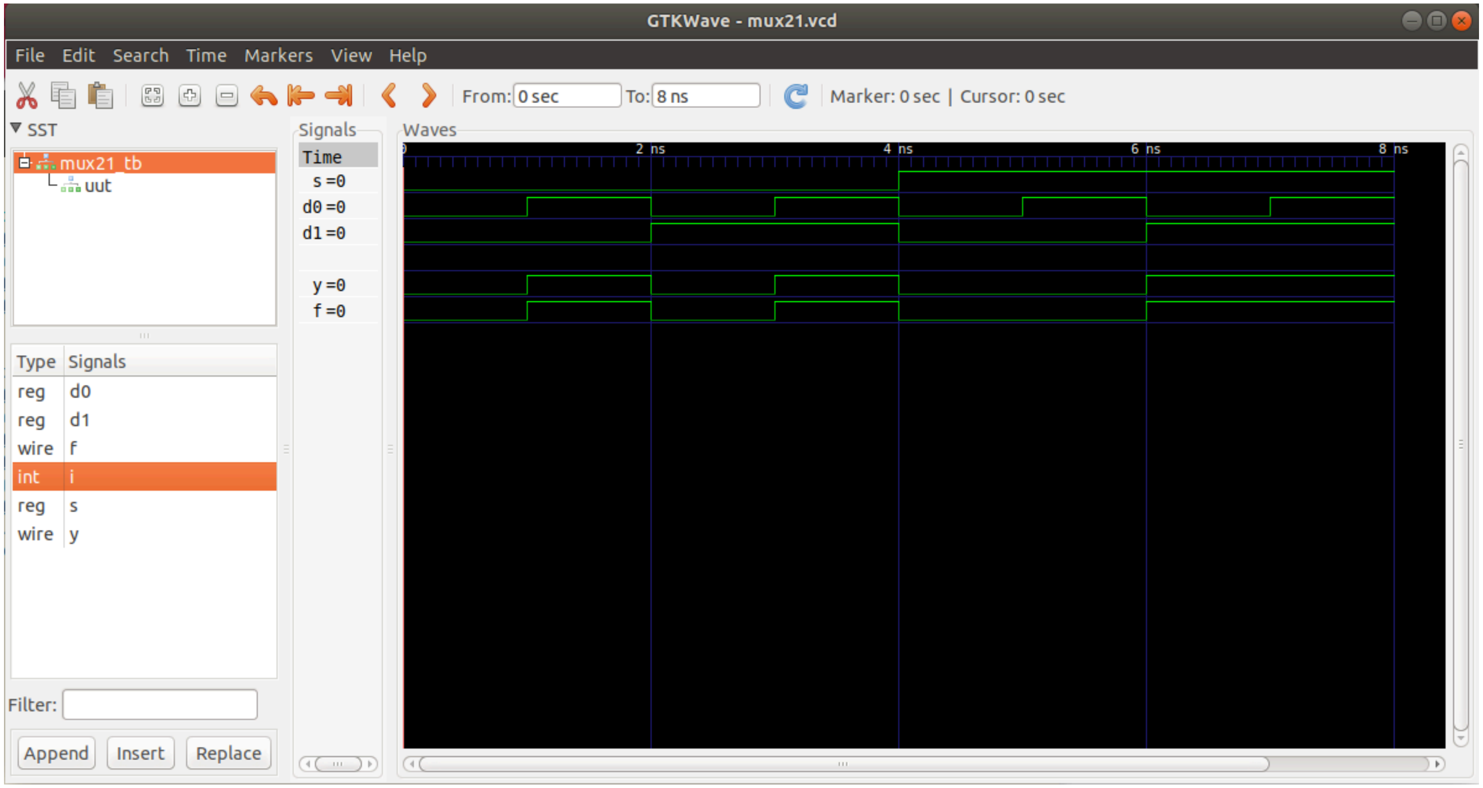
```
// filename: mux21_tb.v
// verify the functionality of mux21.v

// set time tick units and precision to ns
‘timescale 1ns / 1ns

module mux21_tb;

  // UUT inputs
  reg  s;
  reg d0;
  reg d1;
  // UUT outputs
  wire f, y;

  // variables
  integer i;

  // instantiate the unit under test (UUT)
  mux21 uut(
    .s(s),
    .d1(d1),
    .d0(d0),
    .f(f),
    .y(y)
  );

  // initialize inputs
  initial begin
    s = 0;
    d1 = 0;
    d0 = 0;
  end
```

```
  // set time format to be ns
  initial $timeformat(-9, 1, "ns", 10);
  /*
   * $timeformat(unit, precision, "unit", minwidth);
   * unit is the base that time is to be displayed, from 0 to -15 (s to fs)
   * precision is the number of decimal points to display
   * "unit"    is a string appended to the time, such as " ns".
   * minwidth  is the minimum number of characters that will be displayed.
   */

  // output the simulation in graphical format
  initial begin
    $dumpfile("mux21.vcd");
    $dumpvars(0, mux21_tb);
  end

  // generate test patterns
  initial begin
    i = 0;
    repeat (8) begin
      {s, d1, d0} = i;
      #1;
      i = i+1;
    end
    $finish;
  end

  // output the simulation in textual format
  initial begin
    $display("time  s d0 d1  f  y");         // output table: header
    $display("-------------------");         // output table: header
    $monitor("%4d  %1d  %1d  %1d  %1d  %1d", // output table: signal formatting
      $time, s, d0, d1, f, y);              // output table: signals
  end

endmodule
```

# A first look at Verilog (iverilog and GTKwave)

```
$ iverilog –o design mux21.v mux21_tb.v
$ vvp design
$ gtkwave mux21.vcd
```

```
time   s d0 d1  f  y
-------------------
   0   0  0  0  0  0
   1   0  1  0  1  1
   2   0  0  1  0  0
   3   0  1  1  1  1
   4   1  0  0  0  0
   5   1  1  0  0  0
   6   1  0  1  1  1
   7   1  1  1  1  1
```



| Type | Signals |
|---|---|
| reg | d0 |
| reg | d1 |
| wire | f |
| int | i |
| reg | s |
| wire | y |

# A first look at Verilog (Modelsim)

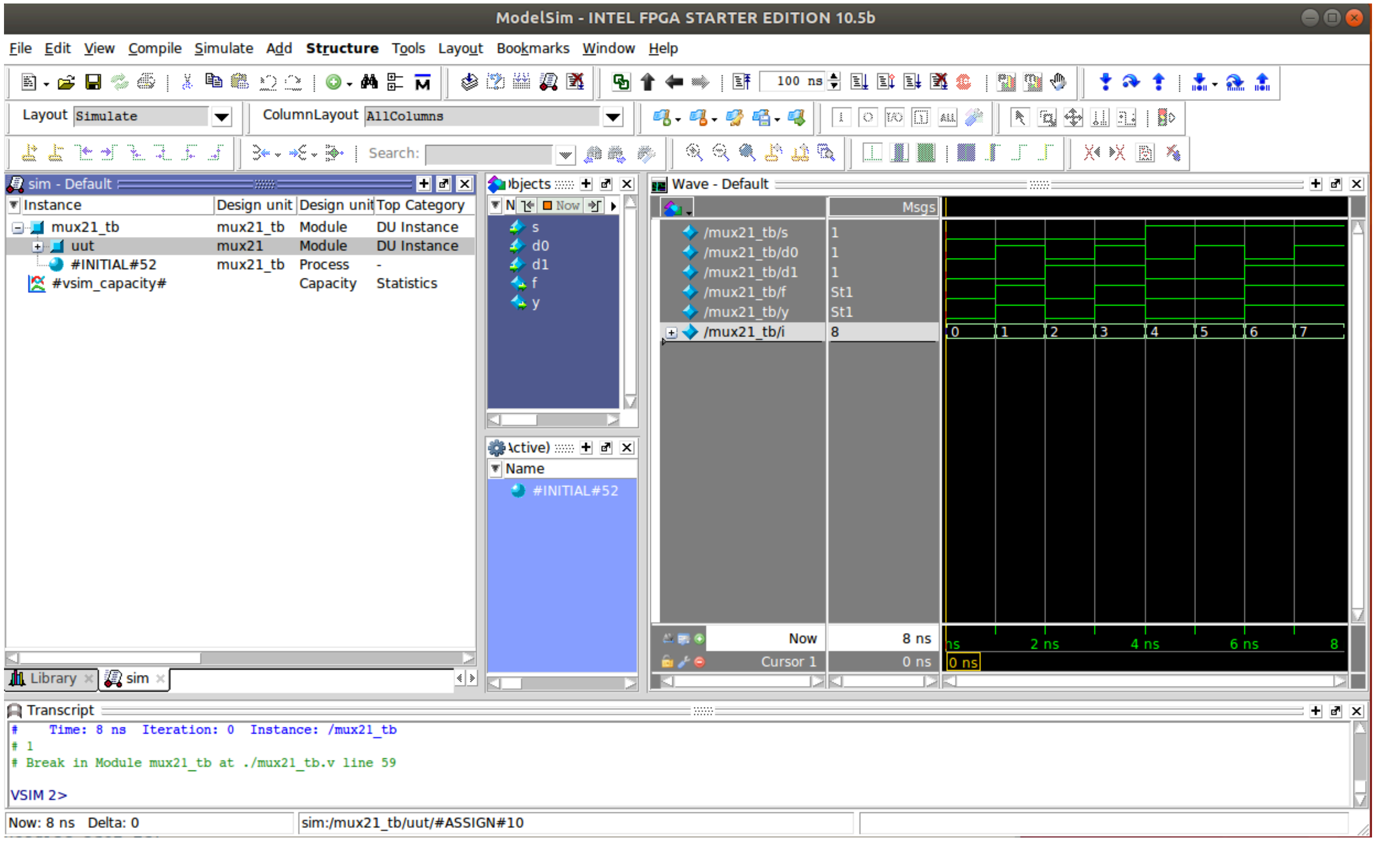
***runmux.scr***

```
#!/bin/bash
vlib work
vlog -work work "./mux21.v"
vlog -work work "./mux21_tb.v"
vsim work.mux21_tb -do simmux.do
```

***simmux.do***

```
restart -f
add wave sim:/mux21_tb/*
run -all
# quit
```

```
$ ./runmux.scr
```


ModelSim - INTEL FPGA STARTER EDITION 10.5b
File Edit View Compile Simulate Add Structure Tools Layout Bookmarks Window Help
100 ns
Search:
sim - Default
Objects
Wave - Default
Msgs
Active)
Name
Library
sim
Transcript

Layout Simulate

ColumnLayout AllColumns

| Instance | Design unit | Design unit | Top Category |
|---|---|---|---|
| mux21_tb | mux21_tb | Module | DU Instance |
| uut | mux21 | Module | DU Instance |
| #INITIAL#52 | mux21_tb | Process | - |
| #vsim_capacity# | | Capacity | Statistics |

Objects: s, d0, d1, f, y

Processes (Active): #INITIAL#52

| Signal | Msgs |
|---|---|
| /mux21_tb/s | 1 |
| /mux21_tb/d0 | 1 |
| /mux21_tb/d1 | 1 |
| /mux21_tb/f | St1 |
| /mux21_tb/y | St1 |
| /mux21_tb/i | 8 |

0
1
2
3
4
5
6
7
ns
2 ns
4 ns
6 ns
8

Now 8 ns

Cursor 1 0 ns

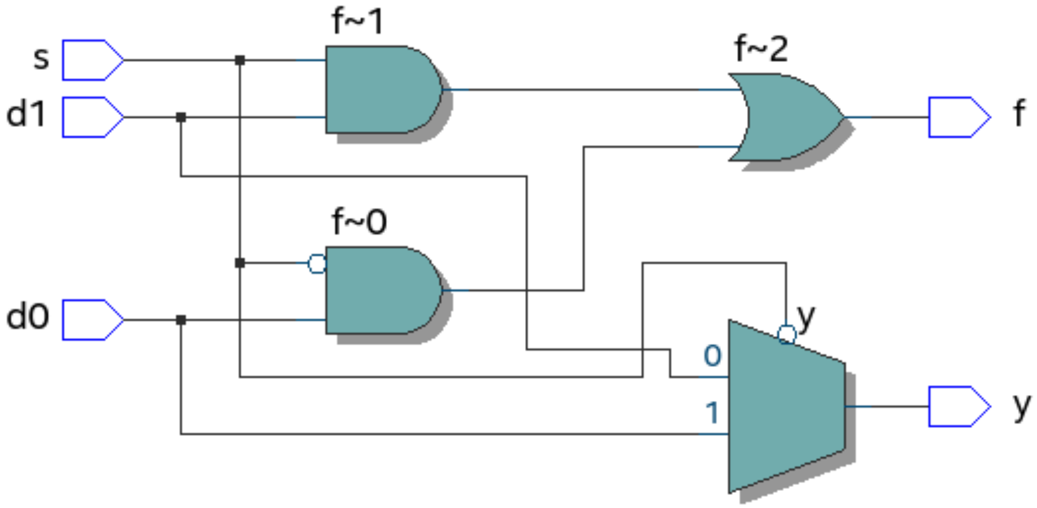
```
#     Time: 8 ns  Iteration: 0  Instance: /mux21_tb
# 1
# Break in Module mux21_tb at ./mux21_tb.v line 59

VSIM 2>
```

Now: 8 ns   Delta: 0   sim:/mux21_tb/uut/#ASSIGN#10

# A first look at Verilog (Quartus)

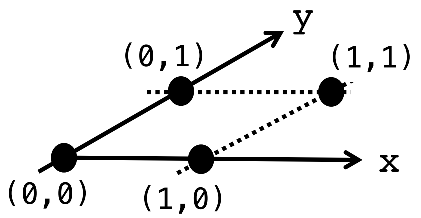
```
$ quartus mux21.qpf
```

*From Verilog code (Boolean equations) to gates*



*Use RTL Viewer to check the circuit synthesized*

# Two-variable Karnaugh Map



*Truth Table*

| x y | f |
|---|---|
| 0 0 | |
| 0 1 | |
| 1 0 | |
| 1 1 | |

*K-Map*

| y \ x | 0 | 1 |
|---|---|---|
| 0 | $m_0$ | $m_2$ |
| 1 | $m_1$ | $m_3$ |

| x y | f |
|---|---|
| 0 0 | 0 |
| 0 1 | 1 |
| 1 0 | 1 |
| 1 1 | 0 |

| y \ x | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

$$f = \bar{x}\cdot y + x\cdot\bar{y} = x \oplus y$$

| x y | g |
|---|---|
| 0 0 | 0 |
| 0 1 | 1 |
| 1 0 | 0 |
| 1 1 | 1 |

| | | x’ | x |
|---|---|---|---|
| | | 0 | 1 |
| y’ | 0 | 0 | 0 |
| y | 1 | 1 | 1 |

$$g = \bar{x}\cdot y + x\cdot y = y\cdot\underbrace{(\bar{x} + x)}_{\textit{combining theorem}} = y$$

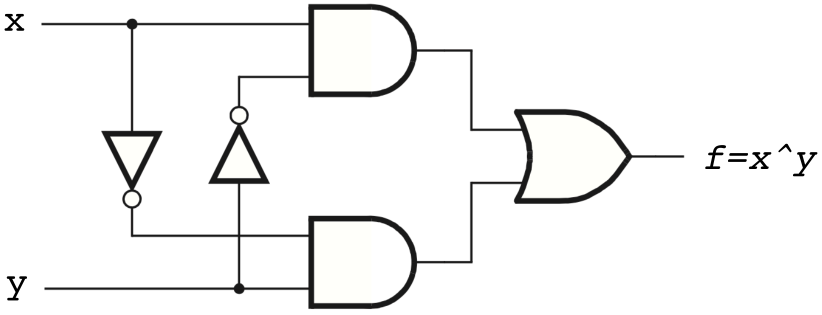
The K-map makes much easier to “visualize” that there is an opportunity to simplify the logic equation (eliminate a variable using the combining theorem)

# Digression: XOR (^) and XNOR (~^) gates

x

| | | 0 | 1 |
|---|---|---|---|
| y | 0 | 0 | 1 |
| | 1 | 1 | 0 |

| x y | f |
|---|---|
| 0 0 | 0 |
| 0 1 | 1 |
| 1 0 | 1 |
| 1 1 | 0 |



SOP implementation of XOR

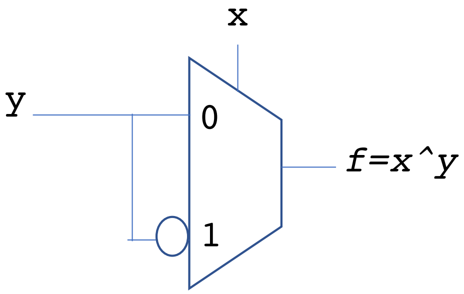
*De Morgan*

$$f = x \oplus y = \bar{x} \cdot y + x \cdot \bar{y} = [(x + \bar{y}) \cdot (\bar{x} + y)]'$$

$$f' = \overline{x \oplus y} = \bar{x} \cdot \bar{y} + x \cdot y = \overline{\bar{x} \cdot y + x \cdot \bar{y}} = (x + \bar{y}) \cdot (\bar{x} + y)$$

*Read the K-Map minterms that are at 0*

*"Flip" the expressions of f*



mux-based implementation of XOR

# Digression: XOR and XNOR gates

| x y | ***f*** |
|---|---|
| 0 0 | 0 |
| 0 1 | 1 |
| 1 0 | 1 |
| 1 1 | 0 |

*XOR*

| x y | ***f'*** |
|---|---|
| 0 0 | 1 |
| 0 1 | 0 |
| 1 0 | 0 |
| 1 1 | 1 |

*XNOR*

$$f = x \oplus y = \bar{x} \cdot y + x \cdot \bar{y}$$

$$f' = \overline{x \oplus y} = \bar{x} \cdot \bar{y} + x \cdot y \triangleq x \odot y$$

The XNOR is also called the ***coincidence*** operator $\odot$

# XOR and XNOR properties

$$f = x \oplus y = \bar{x} \cdot y + x \cdot \bar{y} = [(x + \bar{y}) \cdot (\bar{x} + y)]'$$

$$f' = \overline{x \oplus y} = \bar{x} \cdot \bar{y} + x \cdot y = \overline{\bar{x} \cdot y + x \cdot \bar{y}} = (x + \bar{y}) \cdot (\bar{x} + y)$$

$$\Downarrow$$

$$x \oplus 0 = x \ (= x \cdot 1 + x' \cdot 0)$$

$$x \oplus 1 = x' \ (= x \cdot 0 + x' \cdot 1)$$

$$x \oplus x = 0$$

$$x \oplus x' = 1$$

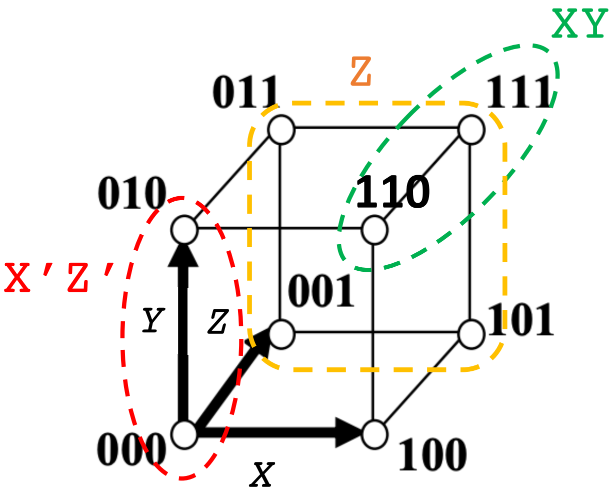
$$x \oplus y = y \oplus x$$

$$x \oplus y \oplus z = (x \oplus y) \oplus z = x \oplus (y \oplus z)$$

$$x(y \oplus z) = xy \oplus xz)$$

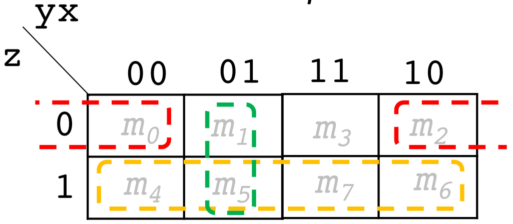
$$x \oplus y' = (x' \cdot y' + x \cdot y'') = \overline{x \oplus y} = x' \oplus y \ (= x' \cdot y' + x'' \cdot y)$$

# Three-variable Karnaugh Map

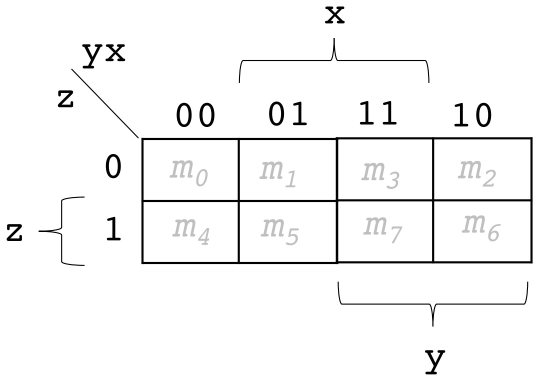


coordinates = (x,y,z)

*K-Map*



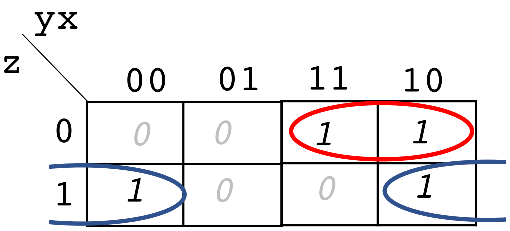
| z \ yx | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | $m_0$ | $m_1$ | $m_3$ | $m_2$ |
| 1 | $m_4$ | $m_5$ | $m_7$ | $m_6$ |



| z \ yx | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | $m_0$ | $m_1$ | $m_3$ | $m_2$ |
| 1 | $m_4$ | $m_5$ | $m_7$ | $m_6$ |

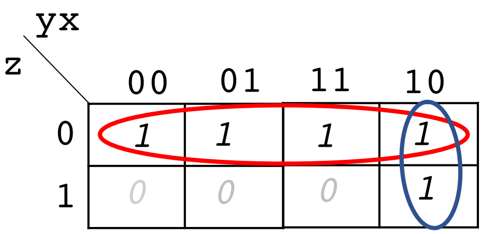
- Clusters of 1 square are minterms
- Clusters of 2 squares eliminate one variable
- Clusters of 4 squares eliminate two variables
- Clusters of 8 squares eliminate three variables
- …

# Logic minimization with K-maps



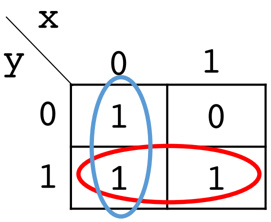
| z yx | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 |

y • z′ + z • x′



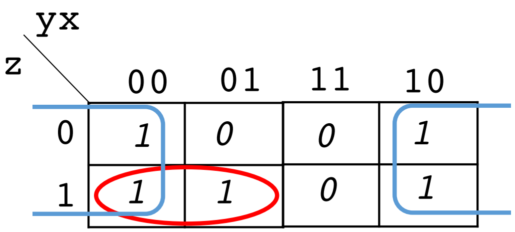
| z yx | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 |

z′ + y • x′



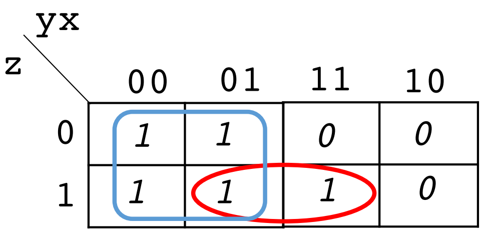
| y x | 0 | 1 |
|---|---|---|
| 0 | 1 | 0 |
| 1 | 1 | 1 |

y + x′



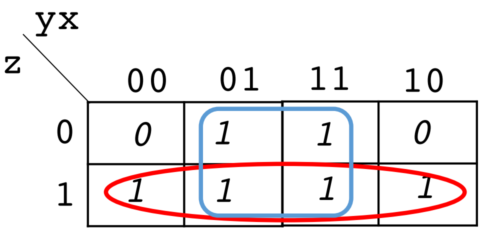
| z yx | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 1 |

z • y′ + x′



| z yx | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 |

z • x + y′



| z yx | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

z + x

# K-map minimization rules

- Each cluster must span a power of 2 (i.e. 1, 2, 4) squares in each direction
- Each cluster must be as large as possible
- A cluster may wrap around the edges of the K-map
- A one in a K-map may be clustered multiple times
- A “don't care” ($d$ *or* $-$) is clustered only if it helps minimize the equation

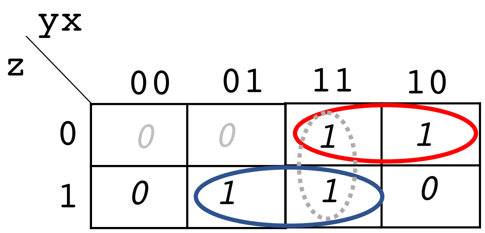
# K-Map definitions (1)

- The basic principle for simplifying SOPs equations is to combine terms using the relationship $P \cdot A + P \cdot A' = P$ (where P may be any product of literals)
- **Implicant** = product of literals
- An equation in SOP form is **minimized** if it uses the least expensive set of implicants
- An implicant is called a **prime implicant** if it cannot be combined with any other implicants to form a new implicant with fewer literals (prime implicants correspond to the largest K-map clusters).
- The implicants in a minimized equation must all be prime implicants, otherwise, they could be combined to reduce the number of literals. However, not all existing prime implicants are needed in forming a minimized equation.

# K-Map definitions (2)

- An essential prime implicant is a prime implicant that cover an output of the function that no combination of other prime implicants is able to cover

*Example*



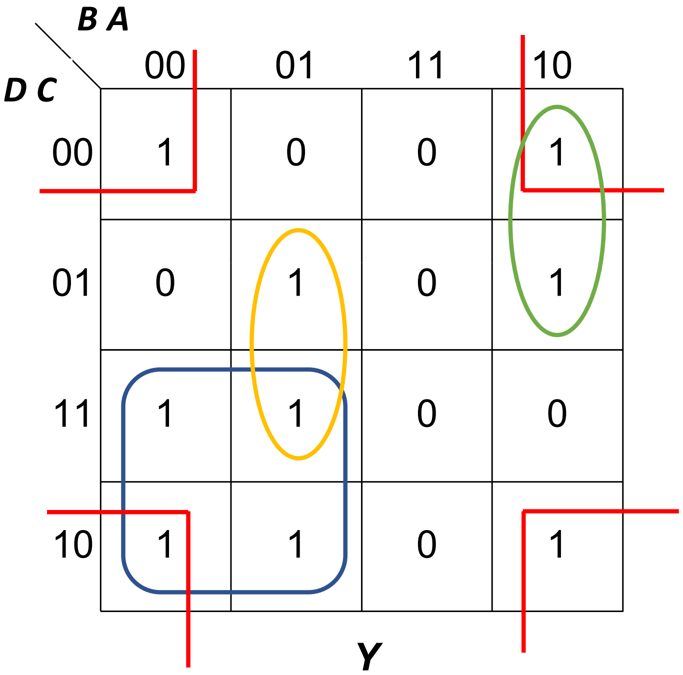
| z \ yx | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 |

$y \cdot z' + x \cdot z$

$x \cdot y$ is a prime implicant, however is not necessary

- Certain functions may have more than one minimized SOP form

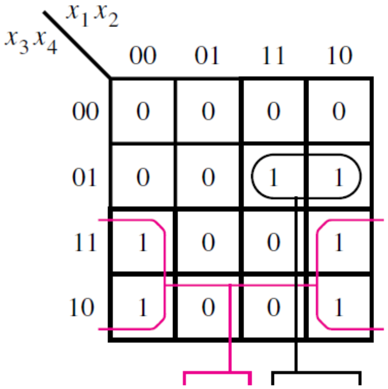
# Four-variable K-Map



| DC \ BA | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 1 | 0 | 0 | 1 |
| 01 | 0 | 1 | 0 | 1 |
| 11 | 1 | 1 | 0 | 0 |
| 10 | 1 | 1 | 0 | 1 |

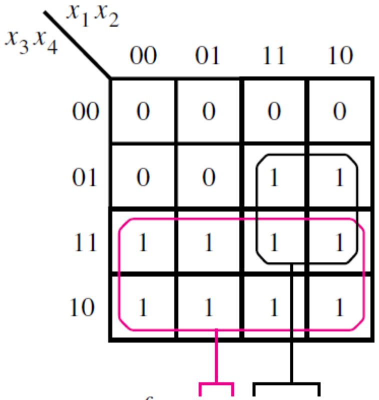
$$Y = C'A' + DB' + CB'A + D'BA'$$

# Four-variable K-Map (few more examples)



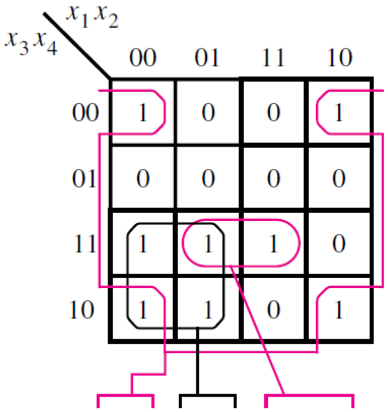
| $x_3x_4 \backslash x_1x_2$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 0 |
| 01 | 0 | 0 | 1 | 1 |
| 11 | 1 | 0 | 0 | 1 |
| 10 | 1 | 0 | 0 | 1 |

$f_1 = \bar{x}_2 x_3 + x_1 \bar{x}_3 x_4$



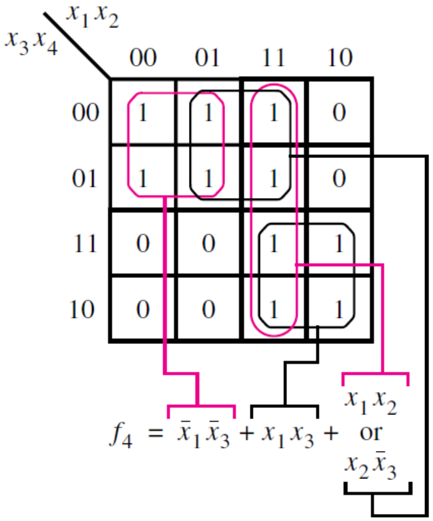
| $x_3x_4 \backslash x_1x_2$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 0 |
| 01 | 0 | 0 | 1 | 1 |
| 11 | 1 | 1 | 1 | 1 |
| 10 | 1 | 1 | 1 | 1 |

$f_2 = x_3 + x_1 x_4$



| $x_3x_4 \backslash x_1x_2$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 1 | 0 | 0 | 1 |
| 01 | 0 | 0 | 0 | 0 |
| 11 | 1 | 1 | 1 | 0 |
| 10 | 1 | 1 | 0 | 1 |

$f_3 = \bar{x}_2 \bar{x}_4 + \bar{x}_1 x_3 + x_2 x_3 x_4$

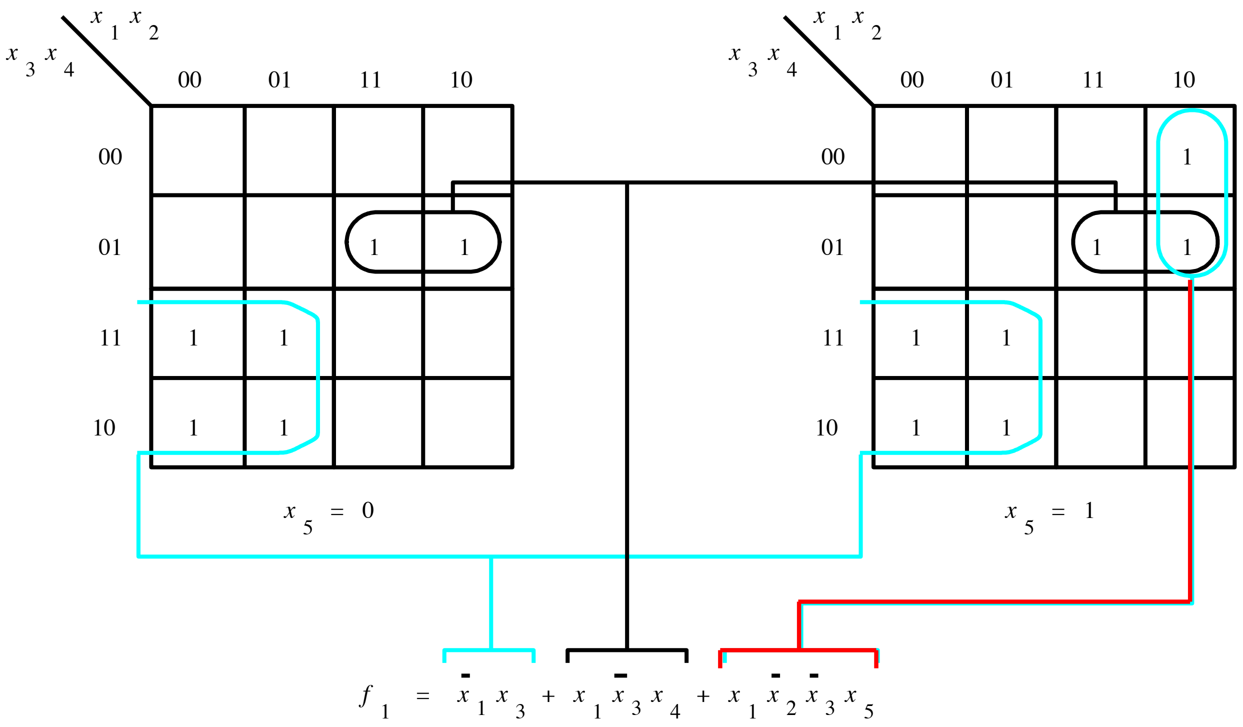


| $x_3x_4 \backslash x_1x_2$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 1 | 1 | 1 | 0 |
| 01 | 1 | 1 | 1 | 0 |
| 11 | 0 | 0 | 1 | 1 |
| 10 | 0 | 0 | 1 | 1 |

$f_4 = \bar{x}_1 \bar{x}_3 + x_1 x_3 + x_1 x_2$ or $x_2 \bar{x}_3$

*The function $f_4$ has two possible SOP minimal forms*

# Five-variable K-Map



$x_5 = 0$

| $x_3x_4$ \ $x_1x_2$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | | | |
| 01 | | | 1 | 1 |
| 11 | 1 | 1 | | |
| 10 | 1 | 1 | | |

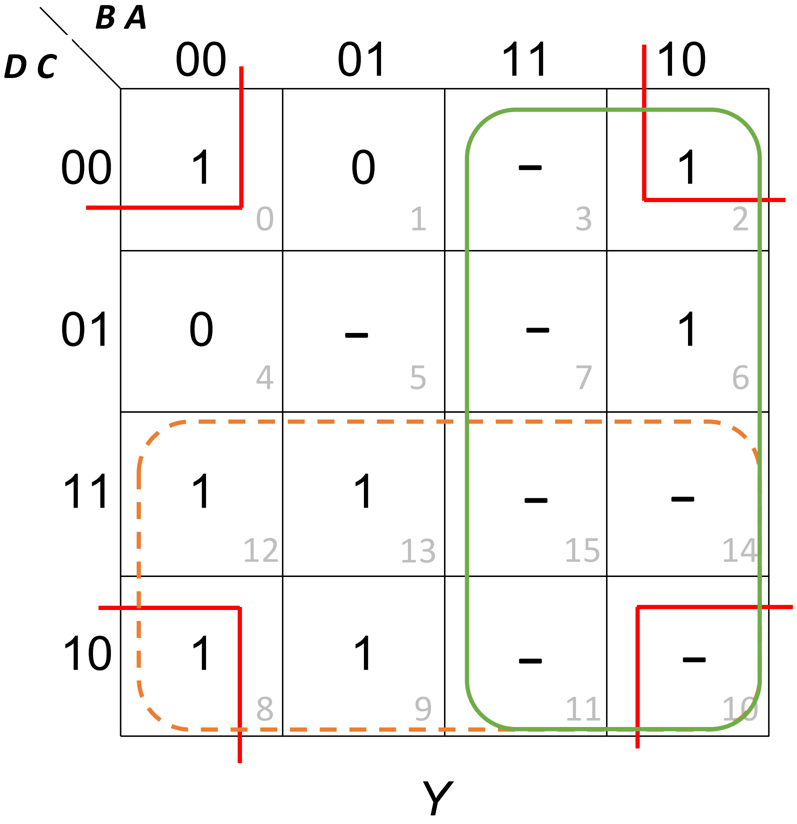
$x_5 = 1$

| $x_3x_4$ \ $x_1x_2$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | | | | 1 |
| 01 | | | 1 | 1 |
| 11 | 1 | 1 | | |
| 10 | 1 | 1 | | |

$$f_1 = \bar{x}_1 x_3 + x_1 \bar{x}_3 x_4 + x_1 \bar{x}_2 \bar{x}_3 x_5$$

*This is nothing else than a “graphical representation” of Shannon Expansion’s Theorem*

# Incompletely specified functions (don’t cares)



| B A<br>D C | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 1<br>0 | 0<br>1 | –<br>3 | 1<br>2 |
| 01 | 0<br>4 | –<br>5 | –<br>7 | 1<br>6 |
| 11 | 1<br>12 | 1<br>13 | –<br>15 | –<br>14 |
| 10 | 1<br>8 | 1<br>9 | –<br>11 | –<br>10 |

*Sometimes the specification of the digital system we want to design guarantees that certain set of input patterns (i.e. minterms) will never be used.*

$$Y = \sum m(0,2,6,8,9,12,13) + \sum d(3,5,7,11,15)$$

$$Y = D + B + \bar{C}\,\bar{A}$$

*NOTE:*
*besides d and – sometimes people have also*
*the unfortunate habit to mark don’t cares with X*

# Map Entered Variables

| z \ yx | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | $w$ | 0 | 1 | 1 |
| 1 | 1 | $w'$ | $w'$ | 1 |

- With more than 5 variables the K-map rapidly becomes unmanageable. Entering variables in the map reduces the required map size, thereby extending K-maps practical usefulness.
- *In general if a variable $P_i$ is placed in square $m_k$ of a map, this means that $F = 1$ when $P_i = 1$* and the variables of the map are chosen so that $m_k = 1$. Given a map with variables $P1$, $P2$, . . ., entered into some of the squares, the minimum sum-of-products (MS) expression for $F$ *is:*

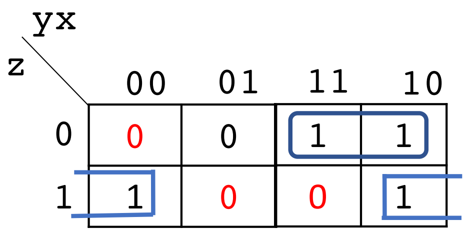
$$F = MS_0 + P_1 \cdot MS_1 + P_2 \cdot MS_2 + \cdots$$

where:

$MS_0$ is the minimum sum obtained by setting $P1 = P2 = \cdots = 0$.
$MS1$ is the minimum sum obtained by setting $P1 = 1$, $Pj = 0$ ($j \neq 1$), and replacing all 1’s on the map with don’t-cares.
$MS2$ is the minimum sum obtained by setting $P2 = 1$, $Pj = 0$ ($j \neq 2$) and replacing all 1’s on the map with don’t-cares.

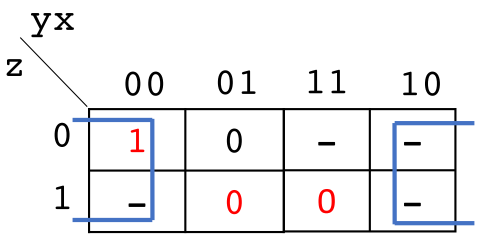
# Map Entered Variables

| z \ yx | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | *w* | 0 | 1 | 1 |
| 1 | 1 | *w'* | *w'* | 1 |



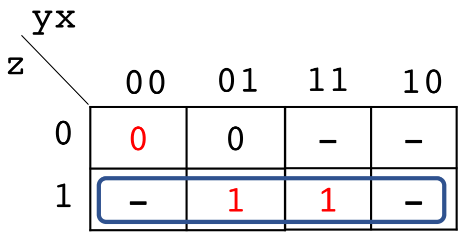
| z \ yx | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 |

$MS_0=z'y+zx'$



| z \ yx | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 1 | 0 | – | – |
| 1 | – | 0 | 0 | – |

$MS_1=wx'$



| z \ yx | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | – | – |
| 1 | – | 1 | 1 | – |

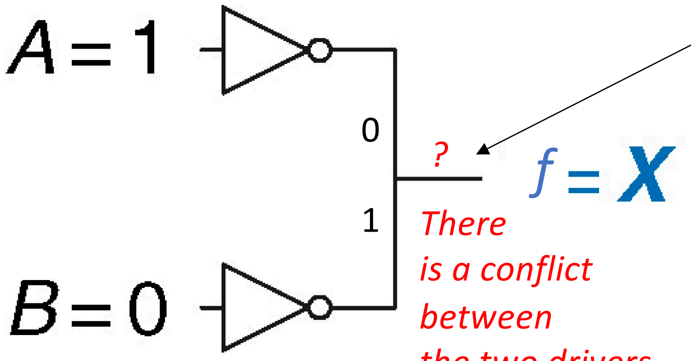
$MS_2=w'z$

$$f = MS_0+MS_1+MS_2 = z'y + \boxed{zx'} + wx' + w'z = z'y + wx' + w'z$$

*zx' is a consensus term w.r.t. wx' and w'z*

# Contention (illegal or indeterminate value X)

*The node has multiple drivers*



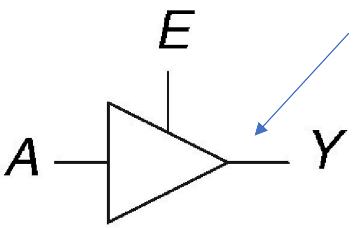
*There is a conflict between the two drivers*

*NOTE: When you see a "X" careful with the meaning: the notation X is over-abused !*
*Sometime X is also used for meaning don't care and sometime for meaning undefined*
*(undefined U is different than indeterminate !)*

# Floating Value ( Z = High Impedance)

*In the rather common case that multiple devices need to share a communication bus (shared interconnection) a solution to avoid contention is to use tri-state buffers.*

**Tri-State Buffer**

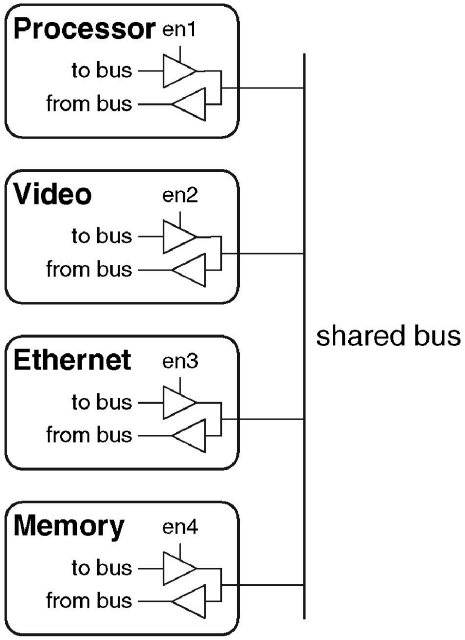


*For E=0 the node Y is left floating*

| E | A | Y |
|---|---|---|
| 0 | 0 | Z |
| 0 | 1 | Z |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

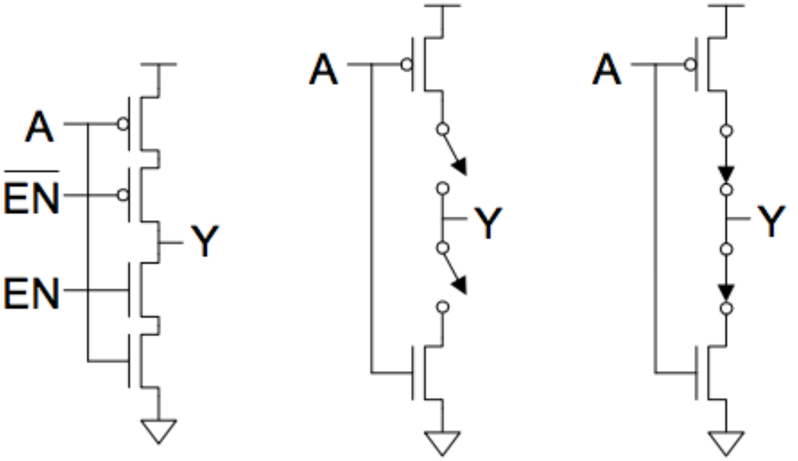
© 2007 Elsevier, Inc. All rights reserved

There are three possible output states: HIGH (1), LOW(0), and floating (Z)



© 2007 Elsevier, Inc. All rights reserved

# Tri-state buffer



EN = 0
Y = 'Z'

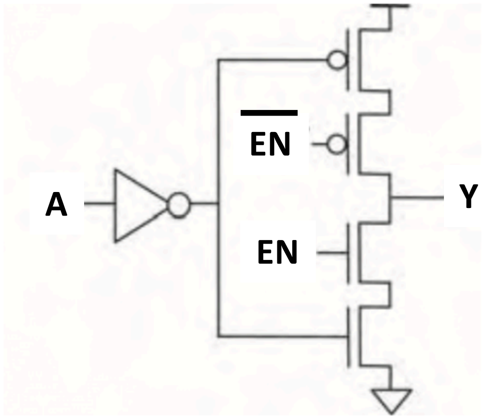
EN = 1
Y = $\overline{A}$

| EN | A | Y |
|---|---|---|
| 0 | 0 | Z |
| 0 | 1 | Z |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

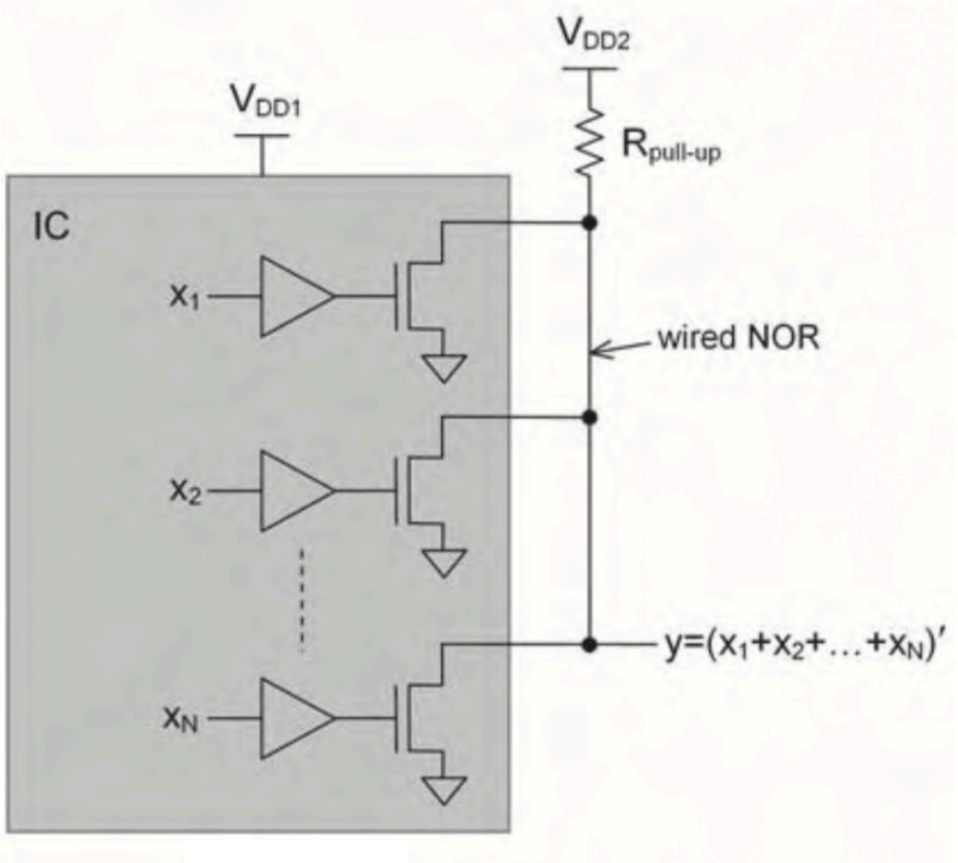
↑

This is **NOT** a tri-state buffer!
It is a tri-state inverter

*Tri-state buffer*



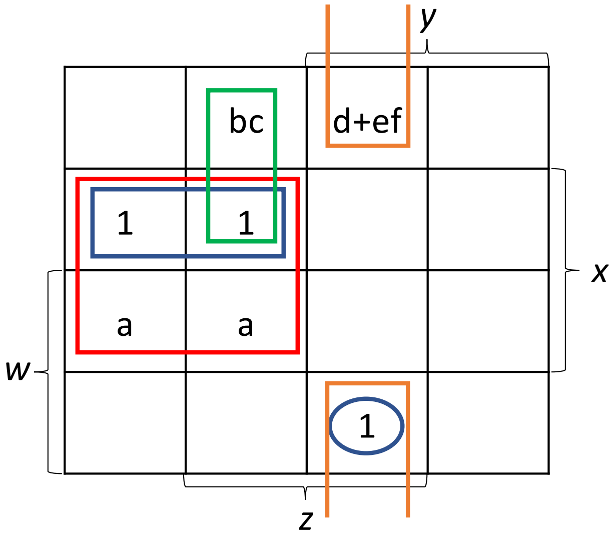
# Open Drain Buffer



# Next Time

- Multiple-output circuits
- Binary Representations and Binary Arithmetic

# Problem



$$MS_0 = w'xy' + wx'yz$$

(a=1) $$MS_1 = axy'$$

(bc=1) $$MS_2 = bcw'y'z$$

(d+ef=1) $$MS_3 = (d + ef)x'yz$$

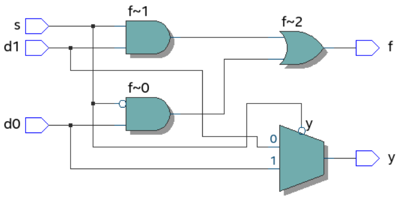
$$F = w'xy' + wx'yz + axy' + bcw'y'z + (d + ef)x'yz$$

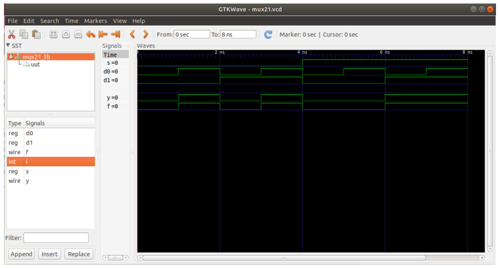
# Binary Representations and Arithmetic

CPEN 230 – Introduction to Digital Logic

# Review: first steps in Verilog

- RTL code for the entry of the circuit (Hardware Implementation)
- Behavioral code the testbench (Verification of Functionality)





| Type | Signals |
|---|---|
| reg | d0 |
| reg | d1 |
| wire | f |
| int | i |
| reg | s |
| wire | y |

```
// filename: mux21.v
// two possible coding styles for implementing a 2:1 multiplexer

module mux21(s,d1,d0,f,y);
  input s, d0, d1;
  output f, y;

  reg y;

  assign f = (~s & d0) | (s & d1);

  always@*
    if (~s)
      y = d0;
    else
      y = d1;

endmodule
```

*blocking assignment*

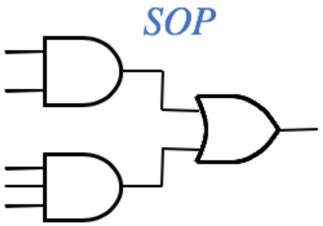
*concurrent coding style*

*procedural coding style*
*(higher level of abstraction)*

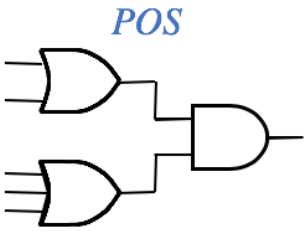
*blocking assignment*

# Review: from equations to gates

- Equations in SOP and POS form result in 2-level logic circuits

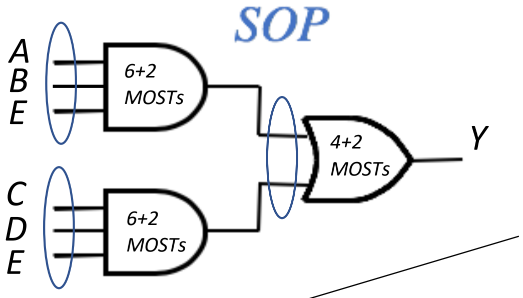


cost = total number of inputs to the logic gates plus number of gates (don't count inversions)



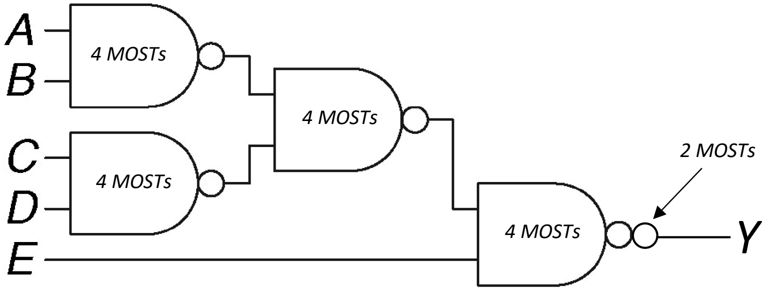
- Minimal SOP and POS are not necessarily the only option nor the "best" option

*cost = 8 inputs + 3 gates = 11*



4-level NAND-NAND circuit

*cost = 8 inputs + 4 gates = 12*



*Minimal Cost ?*

*Facts Checking*

*Total MOSTs: 22*

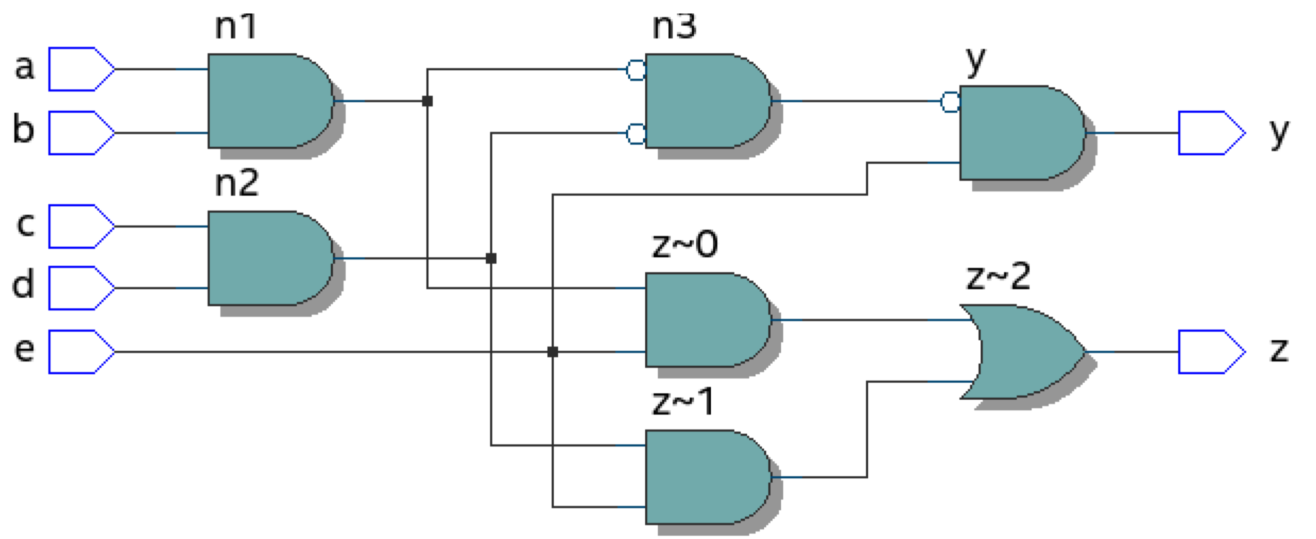
*Total MOSTs: 4×4 + 2 = 18*

*… And, at least for now, let's sweep the issue of speed under the rug!*

# Verilog is your friend !

*… both for enhancing understanding and sparing tedious work*

```
// filename: level3.v
// example showing 4-level logic function vs. SOP

module level3(a,b,c,d,e,y,z);
  input a, b, c, d, e;
  output y, z;

  wire n1, n2, n3;

  assign n1 = ~(a & b);
  assign n2 = ~(c & d);
  assign n3 = ~(n1 & n2);

  assign y = (n3 & e); // 4-level nand-nand
  assign z = (a & b & e) | (c & d & e) ; // sop

endmodule
```

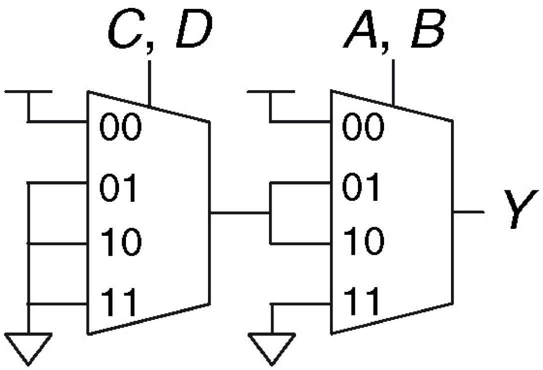


```
row   abcde y z
-------------------
  0   00000 0 0
  1   00001 0 0
  2   00010 0 0
  3   00011 0 0
  4   00100 0 0
  5   00101 0 0
  6   00110 0 0
  7   00111 1 1
  8   01000 0 0
  9   01001 0 0
 10   01010 0 0
 11   01011 0 0
 12   01100 0 0
 13   01101 0 0
 14   01110 0 0
 15   01111 1 1
 16   10000 0 0
 17   10001 0 0
 18   10010 0 0
 19   10011 0 0
 20   10100 0 0
 21   10101 0 0
 22   10110 0 0
 23   10111 1 1
 24   11000 0 0
 25   11001 1 1
 26   11010 0 0
 27   11011 1 1
 28   11100 0 0
 29   11101 1 1
 30   11110 0 0
 31   11111 1 1
```

# Review: from equations to gates

- “Just another brick in the wall”: building logic circuits using muxes

| CDAB | Y |
|---|---|
| 0000 | 1 |
| 0001 | 1 |
| 0010 | 1 |
| 0011 | 0 |
| 0100 | 1 |
| 0101 | 0 |
| 0110 | 0 |
| 0111 | 0 |
| 1000 | 1 |
| 1001 | 0 |
| 1010 | 0 |
| 1011 | 0 |
| 1100 | 1 |
| 1101 | 0 |
| 1110 | 0 |
| 1111 | 0 |

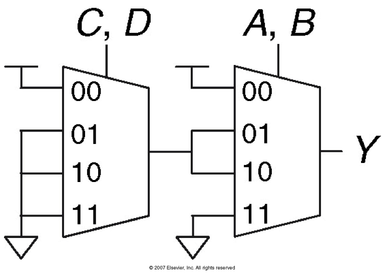


© 2007 Elsevier, Inc. All rights reserved

Expressing Boolean functions in terms of `if-else` statements and `case` statements is usually more “natural” than any other approach.

$$Y = A' \cdot B' + A' \cdot C' \cdot D' + B' \cdot C' \cdot D'$$

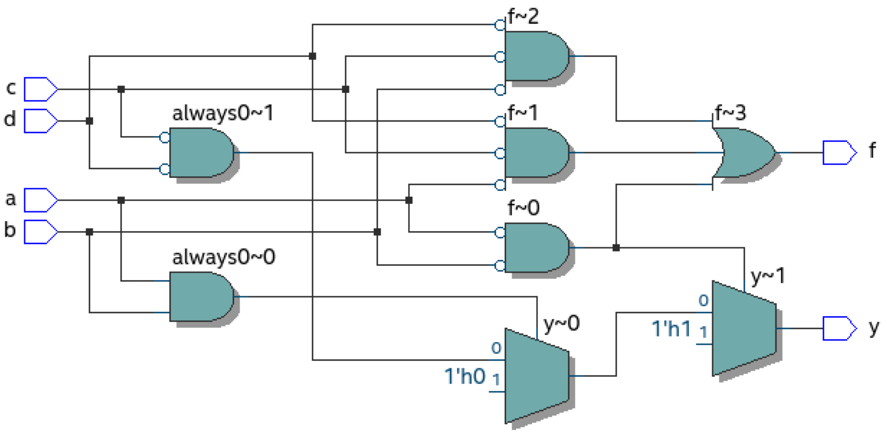
# Building logic circuits using muxes.



```
// filename: lec5.v
// example showing how to build logic functions using "muxes"

module lec5(a,b,c,d,f,y);
  input a, b, c, d;
  output f, y;

  reg y;

  assign f = (~a & ~b) | (~a & ~c & ~d) | (~b & ~c & ~d);

  always @*
    begin
      if (a == 0 && b == 0)
        y = 1;
      else if ( a == 1 && b == 1)
        y = 0;
      else if (c == 0 && d == 0)
        y = 1;
      else
        y = 0;
     end

endmodule
```

*SOP canonical form*

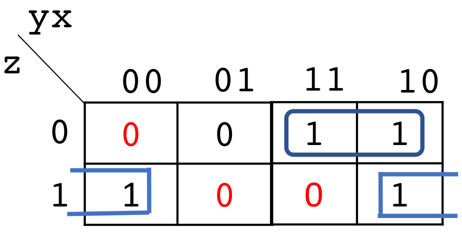
*logical and*



| row | cdab | y | f |
|---|---|---|---|
| 0 | 0000 | 1 | 1 |
| 1 | 0100 | 1 | 1 |
| 2 | 1000 | 1 | 1 |
| 3 | 1100 | 0 | 0 |
| 4 | 0001 | 1 | 1 |
| 5 | 0101 | 0 | 0 |
| 6 | 1001 | 0 | 0 |
| 7 | 1101 | 0 | 0 |
| 8 | 0010 | 1 | 1 |
| 9 | 0110 | 0 | 0 |
| 10 | 1010 | 0 | 0 |
| 11 | 1110 | 0 | 0 |
| 12 | 0011 | 1 | 1 |
| 13 | 0111 | 0 | 0 |
| 14 | 1011 | 0 | 0 |
| 15 | 1111 | 0 | 0 |

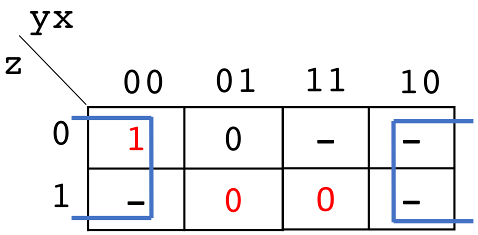
# Review: K-maps and minimization

| z \ yx | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | *w* | 0 | 1 | 1 |
| 1 | 1 | *w'* | *w'* | 1 |



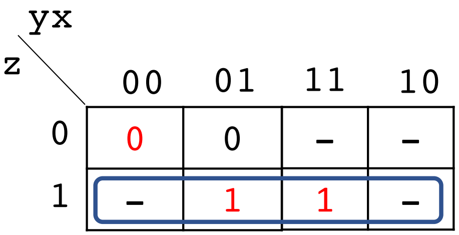
| z \ yx | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 |

$MS_0=z'y+zx'$



| z \ yx | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 1 | 0 | – | – |
| 1 | – | 0 | 0 | – |

$MS_1=wx'$



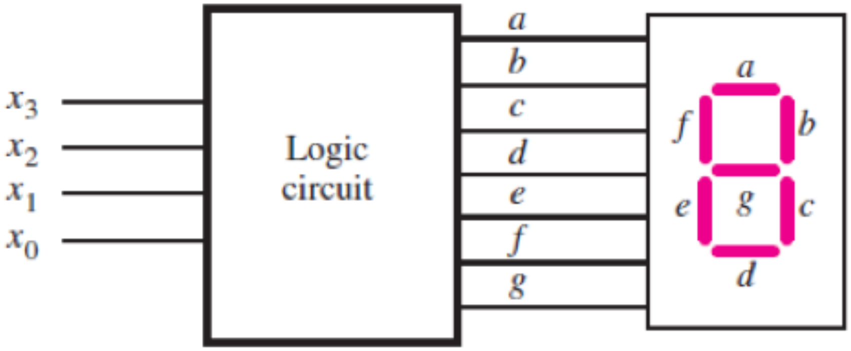
| z \ yx | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | – | – |
| 1 | – | 1 | 1 | – |

$MS_2=w'z$

$$f = MS_0+MS_1+MS_2 = z'y + \boxed{zx'} + wx' + w'z = z'y + wx' + w'z$$

*zx' is a consensus term w.r.t.*
*wx' and w'z*

# Multiple-output Circuits

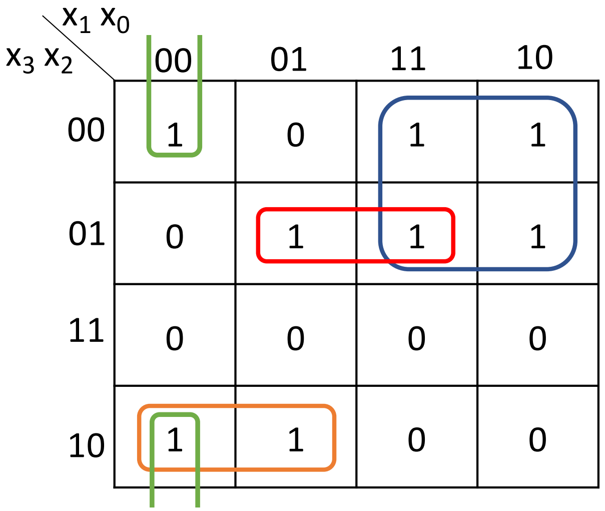


*To design the logic circuit interfacing the 7-segment display we need to write a logic function for each output (a, b, c, d, e, f, g)*

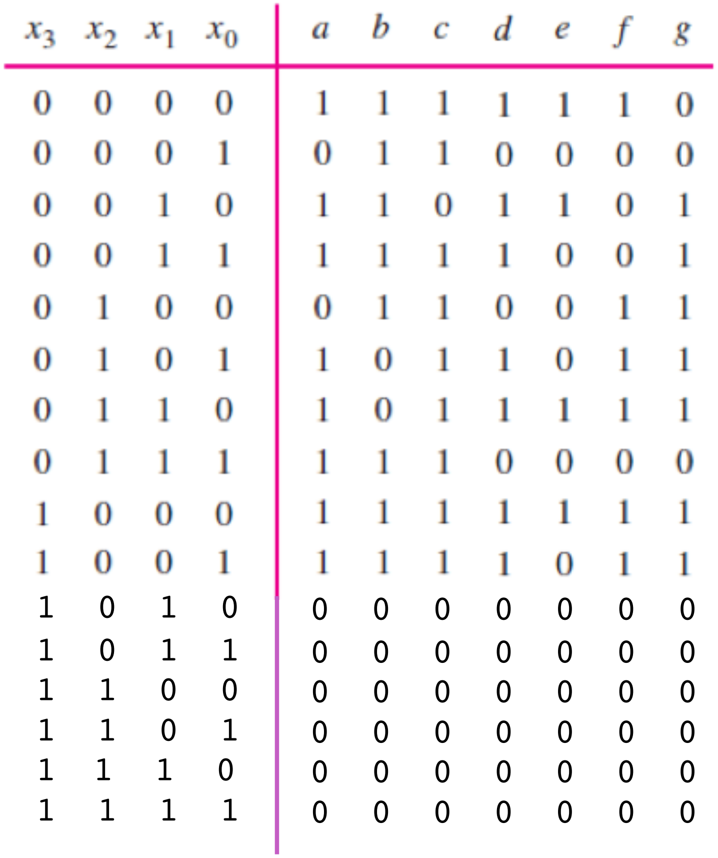
*e.g. write a K-map for each output (a, b, c, d, e, f, g)*

| | $x_3$ | $x_2$ | $x_1$ | $x_0$ | $a$ | $b$ | $c$ | $d$ | $e$ | $f$ | $g$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 3 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| 4 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 5 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 6 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 7 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 8 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 9 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# K-map for output *a*



| $x_3 x_2$ \ $x_1 x_0$ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 1 | 0 | 1 | 1 |
| 01 | 0 | 1 | 1 | 1 |
| 11 | 0 | 0 | 0 | 0 |
| 10 | 1 | 1 | 0 | 0 |

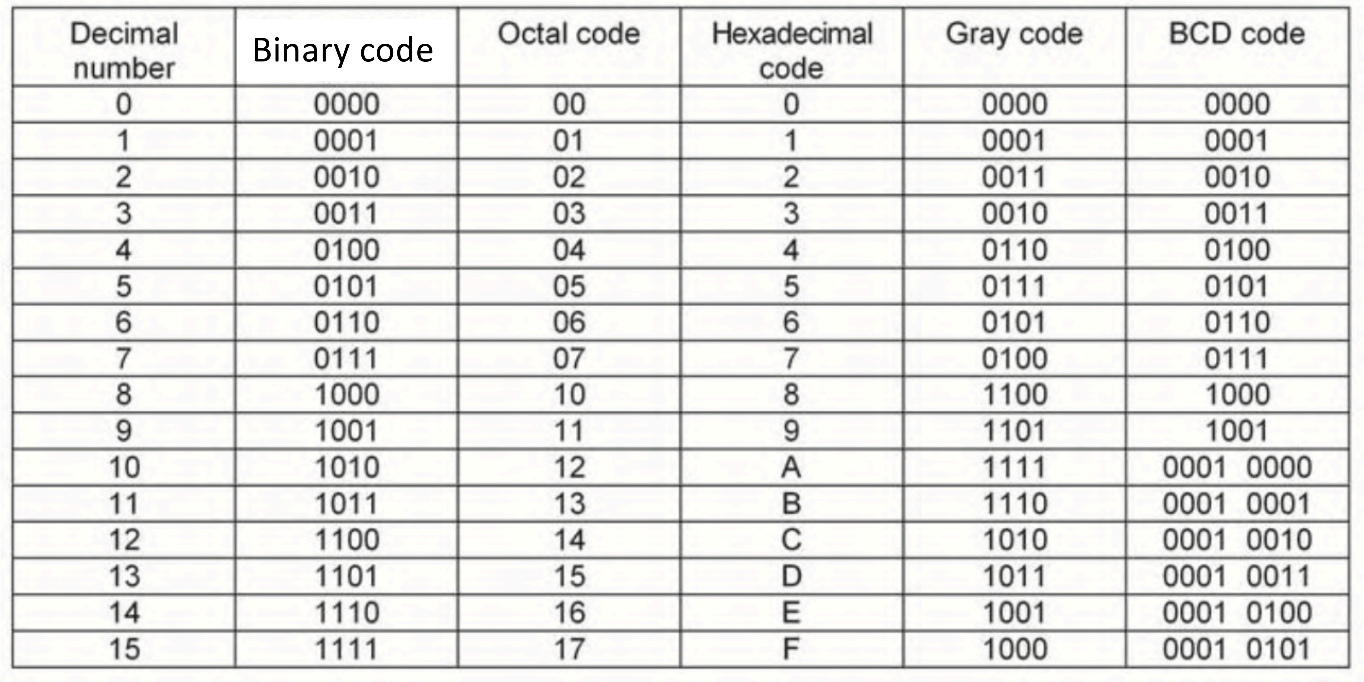


| $x_3$ | $x_2$ | $x_1$ | $x_0$ | $a$ | $b$ | $c$ | $d$ | $e$ | $f$ | $g$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

$$a = \overline{x_3} \cdot x_1 + \overline{x_3} \cdot x_2 \cdot x_0 + x_3 \cdot \overline{x_2} \cdot \overline{x_1} + \overline{x_2} \cdot \overline{x_1} \cdot \overline{x_0}$$

# Binary “patterns” (Codes and Numbers)

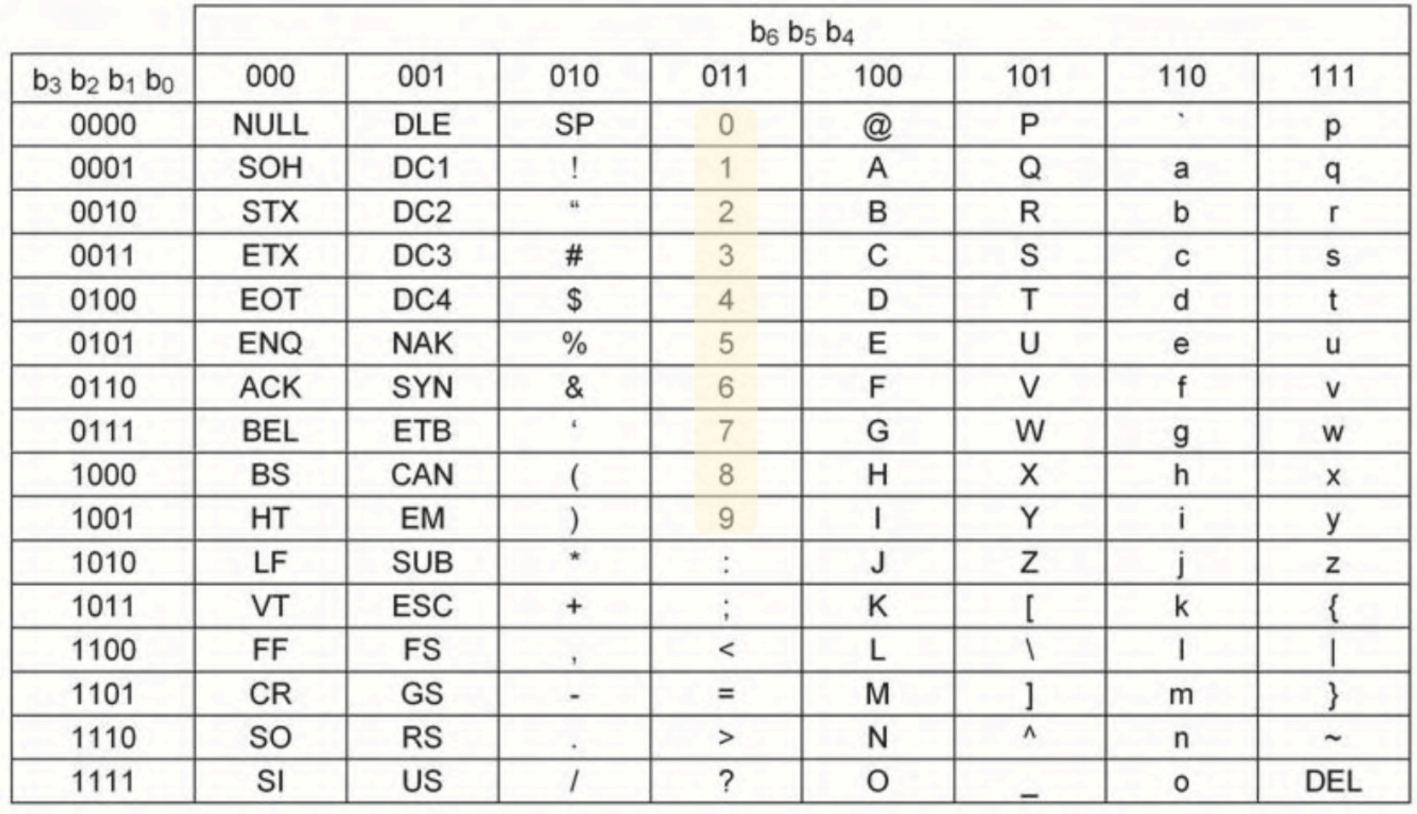
- *Beauty lies in the eye of the Beholder:*
  the same “string” of binary digits (bits) may mean very different things !
  Example: “1111” = $+15_{10}$ or “1111” = $-1_{10}$



| Decimal number | Binary code | Octal code | Hexadecimal code | Gray code | BCD code |
|---|---|---|---|---|---|
| 0 | 0000 | 00 | 0 | 0000 | 0000 |
| 1 | 0001 | 01 | 1 | 0001 | 0001 |
| 2 | 0010 | 02 | 2 | 0011 | 0010 |
| 3 | 0011 | 03 | 3 | 0010 | 0011 |
| 4 | 0100 | 04 | 4 | 0110 | 0100 |
| 5 | 0101 | 05 | 5 | 0111 | 0101 |
| 6 | 0110 | 06 | 6 | 0101 | 0110 |
| 7 | 0111 | 07 | 7 | 0100 | 0111 |
| 8 | 1000 | 10 | 8 | 1100 | 1000 |
| 9 | 1001 | 11 | 9 | 1101 | 1001 |
| 10 | 1010 | 12 | A | 1111 | 0001 0000 |
| 11 | 1011 | 13 | B | 1110 | 0001 0001 |
| 12 | 1100 | 14 | C | 1010 | 0001 0010 |
| 13 | 1101 | 15 | D | 1011 | 0001 0011 |
| 14 | 1110 | 16 | E | 1001 | 0001 0100 |
| 15 | 1111 | 17 | F | 1000 | 0001 0101 |

Codes representing decimal numbers from 0 to 15

# Codes and Numbers



| | $b_6 b_5 b_4$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $b_3 b_2 b_1 b_0$ | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| 0000 | NULL | DLE | SP | 0 | @ | P | ` | p |
| 0001 | SOH | DC1 | ! | 1 | A | Q | a | q |
| 0010 | STX | DC2 | “ | 2 | B | R | b | r |
| 0011 | ETX | DC3 | # | 3 | C | S | c | s |
| 0100 | EOT | DC4 | $ | 4 | D | T | d | t |
| 0101 | ENQ | NAK | % | 5 | E | U | e | u |
| 0110 | ACK | SYN | & | 6 | F | V | f | v |
| 0111 | BEL | ETB | ‘ | 7 | G | W | g | w |
| 1000 | BS | CAN | ( | 8 | H | X | h | x |
| 1001 | HT | EM | ) | 9 | I | Y | i | y |
| 1010 | LF | SUB | * | : | J | Z | j | z |
| 1011 | VT | ESC | + | ; | K | [ | k | { |
| 1100 | FF | FS | , | < | L | \ | l | \| |
| 1101 | CR | GS | - | = | M | ] | m | } |
| 1110 | SO | RS | . | > | N | ^ | n | ~ |
| 1111 | SI | US | / | ? | O | _ | o | DEL |

ASCII code.

# Unsigned Integer Numbers *Range of N-bit numbers:* $[0, 2^N - 1]$

- The most straightforward approach to represent an unsigned number in binary notation is to use the same *positional notation* we have been using with decimal (base 10) numbers since grade school.

$$X = 256_{10} = 2\times10^2 + 5\times10^1 + 6\times10^0$$

*The unsigned integer number X has N=3 digits ($a_2$, $a_1$, $a_0$) and is represented in base (Radix) R=10. In radix 10 using N=3 digits we can represent the numbers in the range [0, 999] =[0, $10^N$–1]*

- In summary (even if we probably never gave to much thought to it) we use a power series expansion.

$$X = (a_{N-1}\, a_{N-2} \cdots\, a_1 a_0)_R = \sum_{i=0}^{N-1} a_i R^i$$

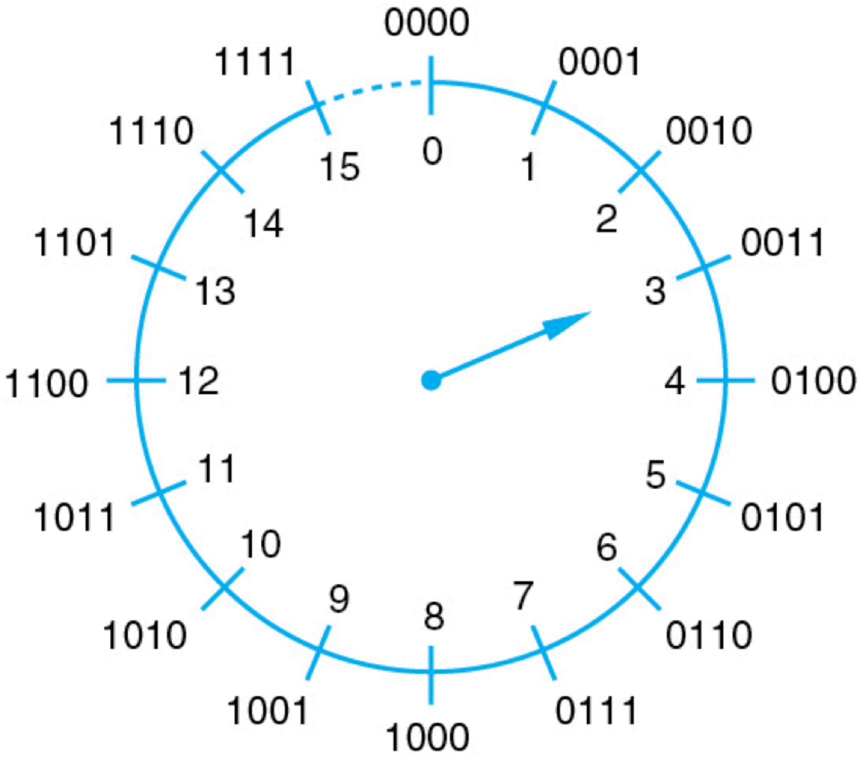
*Where $a_i$ is the coefficient associated to Ri and can take the values in the range $0 \le a_i \le R - 1$*

- So in the case of binary notation the interpretation is as follows:

$$X = 111101_2 = 1\times2^5 + 1\times2^4 + 1\times2^3 + 1\times2^2 + 0\times2^1 + \underset{\substack{\uparrow \\ \textit{must be an odd number}}}{1\times2^0} = 61_{10}$$

*With N=6 binary digits we can represent the numbers in the range [0, $111111_2$] =[0, $63_{10}$]*

# Unsigned Numbers Wheel



# Powers of 2

*To convert between binaries and decimal numbers is useful to memorize the powers of 2 (up to $2^{10}$)*

- $2^0 = 1$
- $2^1 = 2$
- $2^2 = 4$
- $2^3 = 8$
- $2^4 = 16$
- $2^5 = 32$
- $2^6 = 64$
- $2^7 = 128$
- $2^8 = 256$
- $2^9 = 512$
- $2^{10} = 1024$
- $2^{11} = 2048$
- $2^{12} = 4096$
- $2^{13} = 8192$
- $2^{14} = 16384$
- $2^{15} = 32768$

# Large powers of 2

- $2^{10}$ = 1024 = 1 kilo ≈ 1 thousand = $10^3$
- $2^{20}$ = 1,048,576 = 1 mega ≈ 1 million = $10^6$
- $2^{30}$ = 1,073,741,824 = 1 giga ≈ 1 billion = $10^9$

*Memorizing the powers of 2 helps finding the "meaning" of a binary quickly*

<u>*Example:*</u>

$$X = \overset{2^5}{1}111\overset{2^1}{0}\overset{2^0}{1}_2$$

$$111111_2 = 2^6 - 1 = 63$$

$$X = 111101_2 = 63 - 2^1 = 61$$

# Unsigned Integers: Decimal to Binary Conversion

- Two methods:
  - **Method 1:** Find the largest power of 2 that fits, subtract and repeat
  - **Method 2:** Repeatedly divide by 2, remainder goes in next most significant bit

# Unsigned Integers.  Decimal to Binary Conversion Example

**Method 1:** Find the largest power of 2 that fits, subtract and repeat

| | | |
|---|---|---|
| $53_{10}$ | 32×1 | $(32= 2^5)$ |
| 53-32 = 21 | 16×1 | $(16 = 2^4)$ |
| 21-16 = 5 | 4×1 | $(4 = 2^2)$ |
| 5-4 = 1 | 1×1 | $(1 = 2^0)$ |

**$= 110101_2$**

**Method 2:** Repeatedly divide by 2, remainder goes in next most significant bit

| | | | | |
|---|---|---|---|---|
| $53_{10}$ = | 53/2 = 26 | R | 1 (MSB) | |
| | 26/2 = 13 | R | 0 | |
| | 13/2 = 6 | R | 1 | |
| | 6/2 = 3 | R | 0 | |
| | 3/2 = 1 | R | 1 | |
| | 1/2 = 0 | R | 1 (LSB) | **$= 110101_2$** |

# Unsigned Fractional Numbers

- For unsigned fractional binary numbers we continue to use the same notation we have been using for the decimals

$$X = 953.78_{10} = 9\times10^{2} + 5\times10^{1} + 3\times10^{0} + 7\times10^{-1} + 8\times10^{-2}$$

- i.e. an unsigned fractional number X in base R is given by the following power series expansion:

*Fixed Point Notation*

$$X = (a_{N-1}\, a_{N-2}\, \cdots\, a_1 a_0\, .\, a_{-1} \cdots a_{-M})_R = \sum_{i=-M}^{N-1} a_i R^{i}$$

- So in the case of binary notation the interpretation is:

$$X = 1011.01_{2} = 1\times2^{4} + 1\times2^{3} + 0\times2^{2} + 1\times2^{1} + 1\times2^{0} + 0\times2^{-1} + 1\times2^{-2} = 27.25_{10}$$

# Conversion of a fraction from decimal to a binary

- The conversion of a decimal fraction can be done using successive multiplications by the radix R (rather than by successive division by R, as done for the non-fractional part)

$$F = (.\, a_{-1} \cdots a_{-M})_R = \sum_{i=-M}^{-1} a_i R^i = a_{-1} \times R^{-1} + a_{-2} \times R^{-2} + \cdots + a_{-M} \times R^{-M}$$

$$\Downarrow$$

$$F \cdot R = a_{-1} + a_{-2} \times R^{-1} + \cdots + a_M \times R^{-M+1} = a_{-1} + F_1$$

$$F_1 \cdot R = a_{-2} + a_{-3} \times R^{-1} \cdots + a_M \times R^{-M+2} = a_{-2} + F_2$$

$$etc.$$

- This process is continued until we have obtained a “sufficient” number of digits.

# Converting fractions from decimal to binary

*Example*

Repeatedly multiply by 2. The integer part obtained at each step gives the desired digits (starting from the MSB $a_{-1}$ to the LSB)

| | | | |
|---|---|---|---|
| $0.625_{10}$ | 0.625×2 = 1.250 | 1 | ($a_{-1} = 1$) |
| | 0.250**×2** = 0.5 | 0 | ($a_{-2} = 0$) |
| | 0.5 x 2 = 1 | 1 | ($a_{-3} = 0$) |

The process terminates when the product equals 1. However, the process does not always terminate, but if it doesn't, the result is a repeating fraction.

# Converting fractions from decimal to binary

*Example*

| | | | | | |
|---|---|---|---|---|---|
| $0.7_{10}$ | $0.7 \times 2$ | $= 1.4$ | 1 | $(a_{-1} = 1)$ | |
| | $0.4 \times 2$ | $= 0.8$ | 0 | $(a_{-2} = 0)$ | |
| | $0.8 \times 2$ | $= 1.6$ | 1 | $(a_{-3} = 1)$ | |
| | $0.6 \times 2$ | $= 1.2$ | 1 | $(a_{-4} = 1)$ | |
| | $0.2 \times 2$ | $= 0.4$ | 0 | $(a_{-5} = 0)$ | ← Process start  repeating here because 0.4 was previously obtained |

$$0.7_{10} = 0.1\ \underbrace{0110}_{}\ \underbrace{0110}_{}\ \underbrace{0110}_{}\ \ldots_{2}$$

# Addition of unsigned binaries

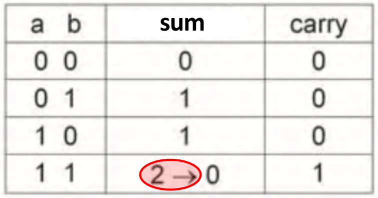
- Decimal

$$\begin{array}{ccccccl} & & 1 & 1 & & & \leftarrow \text{carries} \\ & 3 & 7 & 3 & 4 & & \\ + & 5 & 1 & 6 & 8 & & \\ \hline & 8 & 9 & 0 & 2 & & \end{array}$$

- Binary

$$\begin{array}{ccccccl} & & 1 & 1 & & & \leftarrow \text{carries} \\ & 1 & 0 & 1 & 1 & & \\ + & 0 & 0 & 1 & 1 & & \\ \hline & 1 & 1 & 1 & 0 & & \end{array}$$

*bit-wise addition rules*



| a b | sum | carry |
|---|---|---|
| 0 0 | 0 | 0 |
| 0 1 | 1 | 0 |
| 1 0 | 1 | 0 |
| 1 1 | 2 → 0 | 1 |

# Unsigned Binary Addition Example

- Digital systems operate on a **fixed number of bits**
- Overflow: the result is too big to fit in the available number of bits (with 4 bits we can represent the range [0, 15])

- Add the following 4-bit binary numbers

```
     1
  1001
+ 0101
------
  1110
```

```
    9
+   5     =
------
   14
```

- Add the following 4-bit binary numbers

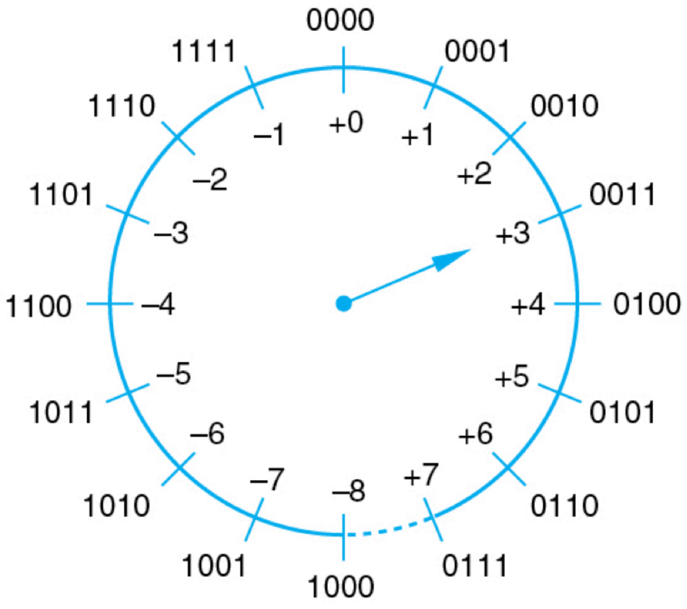
```
   111
  1011
+ 0110
------
 10001
```

Overflow!

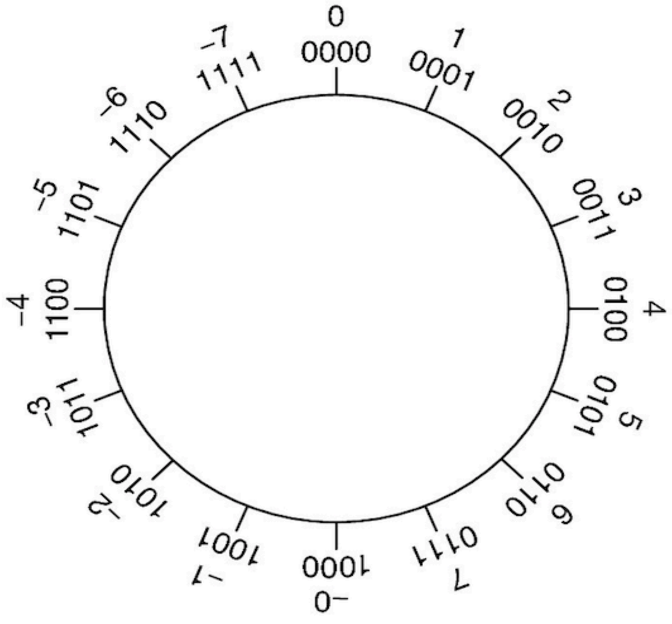
```
   11
+   6     =
------
   17
```

# Signed Integer Numbers

- Sign-Magnitude Notation
- Two’s Complement Notation



*Two’s Complement*



sign-magnitude

We $2^N$–1 binary patterns and we want to use “some” for representing positive quantities and some for negative quantities

# Sign/Magnitude Notation

- Let’s use the most significant digit for the sign and the remaining N–1 digits for the magnitude.
  - and let’s denote positive sign with 0 and negative sign with 1

$$X = (a_{N-1}\ a_{N-2} \cdots\ a_1 a_0)_R = (-1)^{a_{N-1}} \sum_{i=0}^{N-2} a_i R^i$$

- This is pretty much the same approach we have been using with decimal numbers. But … since with decimal numbers we have no limits on the amount of symbols available rather than representing the sign recycling the digits we use the symbol + (to denote positive) and – (to denote negative).

- The range of an N-bit sign/magnitude binary number X is:

$$X \in [-(2^{N-1}-1), (2^{N-1}-1)]$$

# Problems with Sign and Magnitude Notation

- There are two representations for the number 0:

```
1000
0000
```

- Addition doesn't work!

```
+6  =  0110
-6  =  1110
```

```
    6
+ (-6) =
------
    0
```

```
  0110
+ 1110 =
------
 10100     WRONG !
```

*In decimal this would be – $2^2$ = – 4*

# Any other idea ?

- What makes the subtraction challenging is borrowing.
  Can we do a subtraction without ever having to borrow ?

YES WE CAN !

```
 100             99     + 1
- 17  =        - 17
-----          ------------
  83             82     + 1  =  83
```

9 is the greatest digit:
so subtracting from 99
does not require borrows

# Complement-Radix and Complement-Radix-diminished

```
   99     + 1
 - 17
 -----------
   82     + 1  =  83
```

X* = 82 is called the complement-radix-diminished of X=17 -- >
Since in our example R=10 --> **X*=82 is the complement-9 of X=17**

$X^* + X = 99 = R^N – 1$

**X*+X+1 = 100 = R^N**

If our system can only handle up to N=2 digits, then 17+83 = 00 and therefore 83 = 00 – 17 = – 17.
In other words in our system what 83 really represents is the value –17.

$83 = X^* + 1 \triangleq X^\#$ is called the complement radix of X=17
Since in our example R=10 -->
$X^*+1 = X^\# = 83$ is the complement-10 of X=17
$X^*+1+X = X^\#+X = 100 = R^N$

# Further rumination …

Not all the times the number from which we want to subtract is 100.  Can this idea be generalized ?

Y – X

74 – 33 = 41 Easy: no borrow is needed

74 – 36 = 38 Not so easy, because a borrow is needed

$$74 - 36 = 74 + 100 - 100 - 36 = 74 + (100 - 36) - 100 = 74 + \underbrace{(99 + 1 - 36)}_{K_{10} \text{ of } X=36} - 100 = 74 + \overbrace{\underbrace{(99 - 36)}_{K_9 \text{ of } X=36}}^{easy\ =\ 63} + 1 - 100$$

$$K_{10} = K_9 + 1$$

For an N digit number X in base R its R-complement is defined as $K_R = R^N - X$,
while its R-diminished-complement is defined as $K_{R-1} = (R^N - 1) - X$

Thus the required decimal subtraction (74 – 36) can be performed by addition of the 10-complement of 36 and deletion of the leading digit:

74 – 36 = 74 + 64 – 100 = 138 – 100 = 38

The subtraction 138 – 100 is trivial because it implies ignoring the leading digit (i.e. the carry-out $R^N$) in 138

# Further Rumination …

Does this really work all the time even when X < Y ( in our previous example we used X ≥ Y) ?

*Example 1: N = 3*

*X = 045, Y=027*

$$
\begin{aligned}
X - Y &= 045 - 027 \\
&= 045 + 1000 - 1000 - 027 \\
&= 045 + (999 - 027) + 1 - 1000 \\
&= 045 + 972 + 1 - 1000 \\
&= 1018 - 1000 \\
&= 018
\end{aligned}
$$

*Example 2: N = 3*

*X = 027, Y=045*

$$
\begin{aligned}
X - Y &= 027 - 045 \\
&= 027 + 1000 - 1000 - 045 \\
&= 027 + (999 - 045) + 1 - 1000 \\
&= 027 + 954 + 1 - 1000 \\
&= 982 - \cancel{1000}
\end{aligned}
$$

*The carry-out (= $10^3$) can be discarded*

X – Y = 982 – 1000
982 = X – Y + 1000
982 = 1000 – (Y – X)

982 is the negative number that results when forming the 10's complement of Y – X = 018. Thus the number 982 is the 10's complement representation of –18.

# The Two’s Complement notation

- In the 2’s complement notation a positive number X is represented by a 0 followed by the magnitude as in the sign and magnitude notation, however, a negative number –X, is represented by its 2-complement $X^{\#}$. If a positive number X is represented with N bits, its 2-complement is defined as the word $X^{\#}$ of length N-bits given by:

X# = complement-2 of X = complement-1 of X + 1

$$\boxed{X^{\#}} = \underbrace{2^N - X}_{256-97=\ 159} = 2^N - X - 1 + 1 = \boxed{2^N - 1 - X} + 1 = \boxed{X^{*} + 1}$$

X# = complement-2 of X = 2^N - X

X* = complement-1 of X = (2^N - 1 - X)

*Example: (N=8)* **Find the Complement-2 of X = 97**

$2^N{-}1 \;=\; 11111111$

$X \;=\; 01100001$

*X “flipped” (= not X)*

$2^N{-}1{-}X \;=\; 10011110 \;=\; X^{*} \;=\; X'$ ← **X* = complement-1 of X ( = 2^N - 1 - X)**

$+ \quad 1$

$X^{\#} \;=\; \underbrace{10011111}_{2^7 + 2^5 - 1\ =\ 128+31\ =\ 159}$

**X# = complement-2 of X ( = 2^N - X)**

$X \;=\; 97_{10}$

$X^{\#} \;=\; -2^7+(2^5-1) = -128+31 = -97_{10}$

# Two’s Complement Numbers

- Don’t have same problems as sign/magnitude numbers:
  - **Addition works**
  - **There is a single representation for 0**

# Two’s complement integer numbers

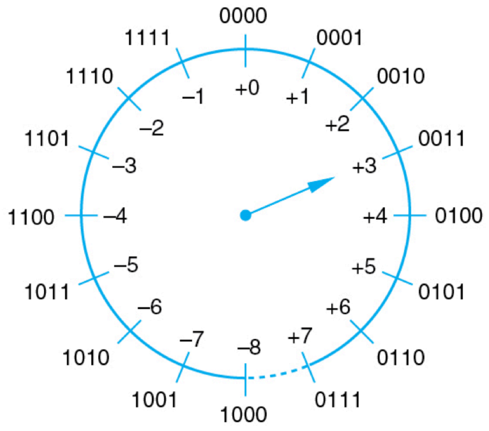
- MSB has value of $-2^{N-1}$

$$X = a_{N-1}(-2^{N-1}) + \sum_{i=0}^{N-2} a_i \times 2^i$$

- The most significant bit still indicates the sign (1 = negative, 0 = positive)
- Range of an *N*-bit two’s complement number:

$$[-2^{N-1}, +2^{N-1} - 1]$$

*Two’s Complement*



*Most positive 4-bit number: 0100*

*Most negative 4-bit number: 1000*

# Taking the Two’s Complement

- **Method:**
  1. “Flip” (invert) the bits
  2. Add 1

**Example:** Flip the sign of $3_{10} = 0011_2$

```
1. 1100 ←—— invert the bits
2. +   1
   ————
    1101  = -3₁₀
```

$$\underbrace{1101}_{-1\times2^3 + 1\times2^4 + 1\times2^0 = -8+4+1=-3} = -3_{10}$$

# Two’s Complement Examples

- Take the two’s complement of $6_{10} = 0110_2$

  1. 1001
  2. + 1

  $1010_2 = -6_{10}$

- What is the decimal value of the two’s complement number $1001_2$? $(-8+1= - 7)$

  1. 0110
  2. + 1

  $0111_2 = 7_{10}$ --> so $1001_2 = -7_{10}$

# Two’s Complement Addition

*If the two operands are of opposite sign the result is always within the range of numbers that can be represented*
4 bits range: [−8, 7]

- Add 6 + (-6) using two’s complement numbers

```
          111
           0110
      +    1010
      ----------
          10000
discard ↗
```

- Add -2 + 3 using two’s complement numbers

```
          111
           1110
      +    0011
      ----------
          10001
discard ↗
```

# Two’s Complement Addition

*If adding two operands of the same sign the sign of the result comes out different there is an error (overflow)*

- Add 6 + 6 using two’s complement numbers

```
     11
     0110
 +   0110
 --------
     1100
```

12 is out of the range we can represent with 4 bits (4-bit range [−8, 7])

*If we had an extra bit the result would have been just fine (00110+00110 = 01100)*

*ERROR*

- Add **−6** – 3 using two’s complement numbers

```
     1
      1010
 +    1101
 --------
     10111
```

−9 is out of the range we can represent with 4 bits (4-bit range [−8, 7])

*If we could have used an extra bit the result would have been just fine*

*11010 + 11101 = 110111*

*discard*

*ERROR*

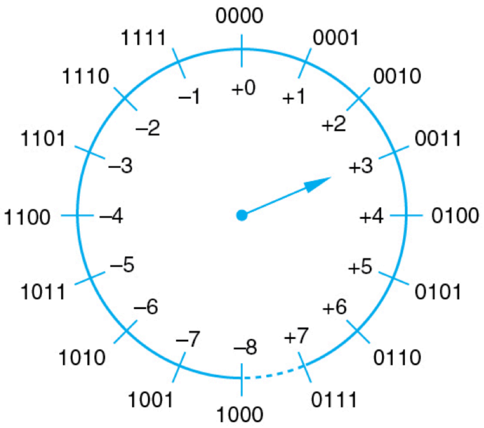
# Increasing Bit Width

Extend number from *N* to *M* bits (*M* > *N*) :

- Sign-extension
- Zero-extension

# Sign Extension

- Sign bit copied to MSB's
- Number value is same
- Example 1:
  - 4-bit representation of 3 = 0011
  - 8-bit sign-extended value: 00000011
- Example 2:
  - 4-bit representation of −5 = 1011
  - 8-bit sign-extended value: 11111011



# Zero Extension

- Zeros copied to MSB's
- Value changes for negative numbers
- Example 1:
  - 4-bit value = 0011 = $3_{10}$
  - 8-bit zero-extended value: 00000011 = $3_{10}$
- Example 2:
  - 4-bit value = 1011 = $-5_{10}$
  - 8-bit zero-extended value: 00001011 = $11_{10}$

# Signed Fractional Numbers

*Example*

$-\ 5.75_{10}$

First, find the binary representation of (positive) $5.75_{10}$

$5.75_{10} = 0101.1100_2$

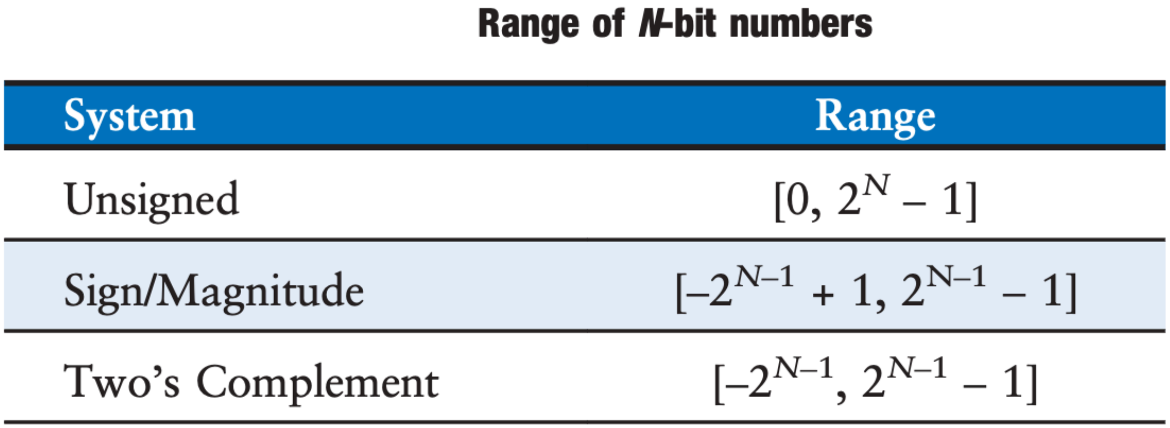
Then convert it to negative using complement 2 notation (as usual invert all bits and add 1 in the LSB position)

$$0101.1100_2 \longrightarrow \begin{array}{r} 1010.0011_2 \\ +\ 0000.0001_2 \\ \hline 1010.0100_2 \end{array} \Longleftrightarrow -2^3 + 2^1 + 2^{-2} = -8 + 2 + 0.25 = 5.75_{10}$$

$$X = a_{N-1}(-2^{N-1}) + \sum_{i=-M}^{N-2} a_i \times 2^i$$

*Signed Binary Numbers (i.e.* $K_2$ *Number System)*
(Fixed point Notation)

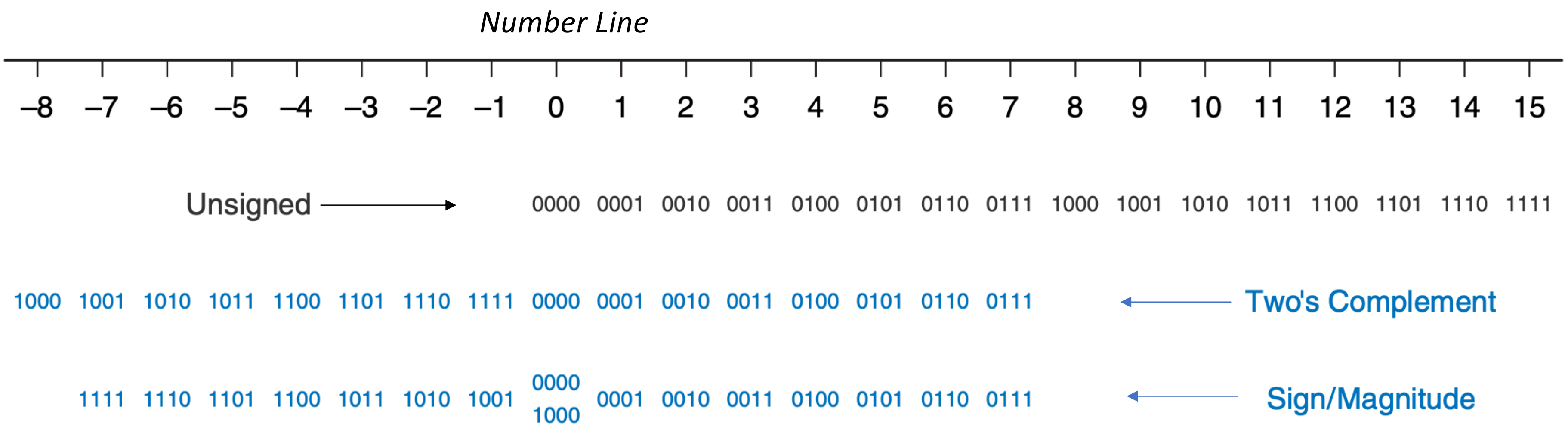
# Comparison of Binary Systems



**Range of *N*-bit numbers**

| System | Range |
|---|---|
| Unsigned | $[0, 2^{N} - 1]$ |
| Sign/Magnitude | $[-2^{N-1} + 1, 2^{N-1} - 1]$ |
| Two's Complement | $[-2^{N-1}, 2^{N-1} - 1]$ |

# Comparison of Binary Numbers

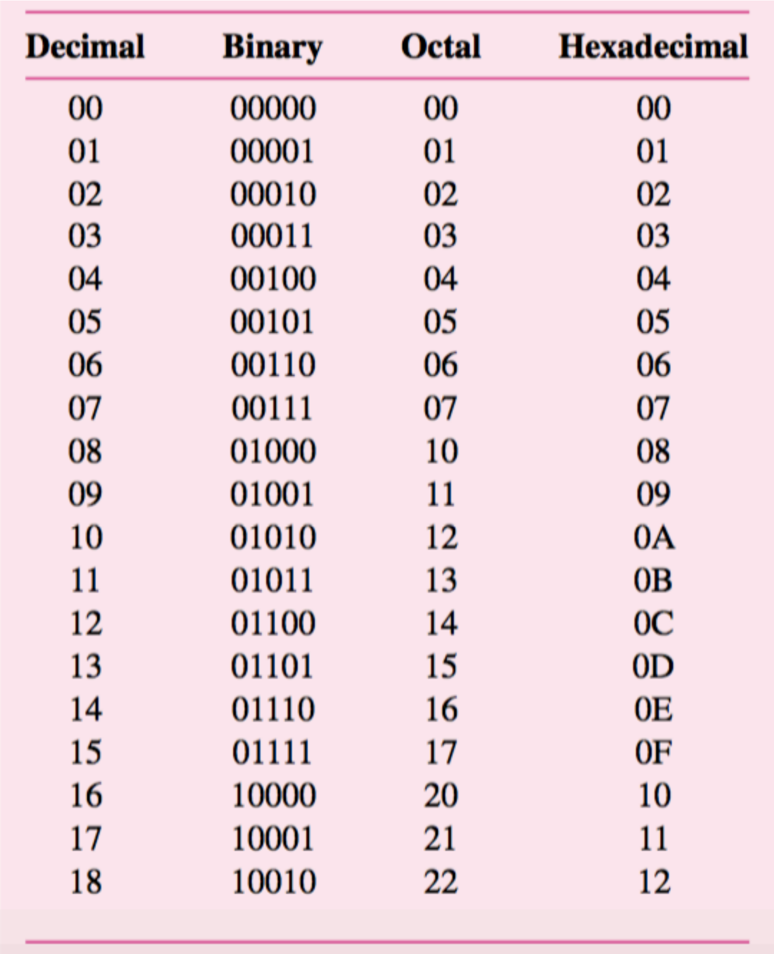


| *Number Line* | –8 | –7 | –6 | –5 | –4 | –3 | –2 | –1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Unsigned → |  |  |  |  |  |  |  |  | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 |
| ← Two's Complement | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 |  |  |  |  |  |  |  |  |
| ← Sign/Magnitude |  | 1111 | 1110 | 1101 | 1100 | 1011 | 1010 | 1001 | 0000<br>1000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 |  |  |  |  |  |  |  |  |

**Number line and 4-bit binary encodings**

# Binaries vs. Octals and Hexadecimals

Binary = base 2
Octal = base 8
Hexadecimal = base 16

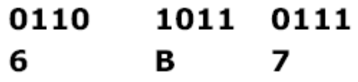


| Decimal | Binary | Octal | Hexadecimal |
|---|---|---|---|
| 00 | 00000 | 00 | 00 |
| 01 | 00001 | 01 | 01 |
| 02 | 00010 | 02 | 02 |
| 03 | 00011 | 03 | 03 |
| 04 | 00100 | 04 | 04 |
| 05 | 00101 | 05 | 05 |
| 06 | 00110 | 06 | 06 |
| 07 | 00111 | 07 | 07 |
| 08 | 01000 | 10 | 08 |
| 09 | 01001 | 11 | 09 |
| 10 | 01010 | 12 | 0A |
| 11 | 01011 | 13 | 0B |
| 12 | 01100 | 14 | 0C |
| 13 | 01101 | 15 | 0D |
| 14 | 01110 | 16 | 0E |
| 15 | 01111 | 17 | 0F |
| 16 | 10000 | 20 | 10 |
| 17 | 10001 | 21 | 11 |
| 18 | 10010 | 22 | 12 |

Octals and Hexadecimals are just a short-hand notation of Binaries

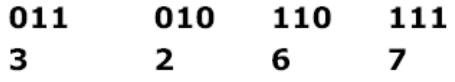
# Binary/Octal/Hexadecimal conversions

Binary-to-hexadecimal: collect binary digits into groups of four (nibble) and assign each group a hexadecimal digit



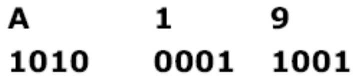
| | | |
|---|---|---|
| 0110 | 1011 | 0111 |
| 6 | B | 7 |

Binary-to-octal: Collect binary digits into groups of three and convert each group to the corresponding octal digit



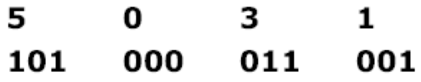
| | | | |
|---|---|---|---|
| 011 | 010 | 110 | 111 |
| 3 | 2 | 6 | 7 |

Hexadecimal-to-binary: Convert each hexadecimal digit to the corresponding binary nibble



| | | |
|---|---|---|
| A | 1 | 9 |
| 1010 | 0001 | 1001 |

Octal-to-binary: Convert each octal digit to the corresponding group of three binary digits



| | | | |
|---|---|---|---|
| 5 | 0 | 3 | 1 |
| 101 | 000 | 011 | 001 |

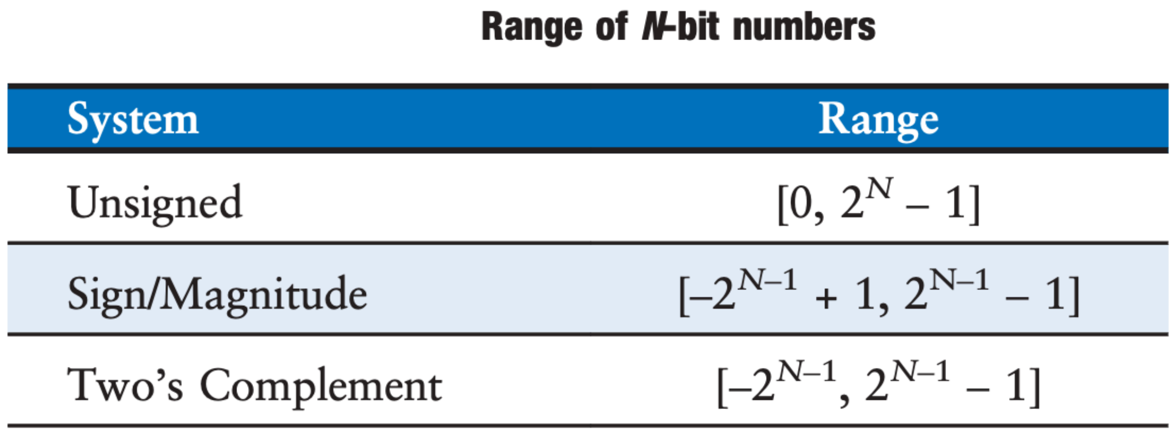
# Next Time

- C.L. Building Blocks
- Gate Delays

# C.L. Building Blocks and Gate Delays

CPEN 230 – Introduction to Digital Logic

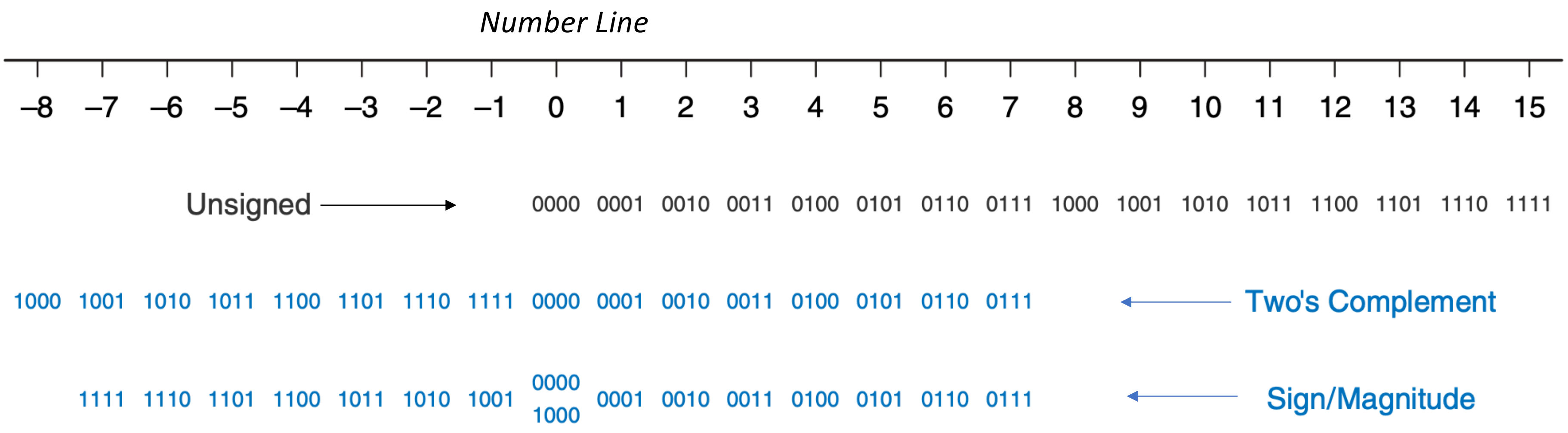
# Review: Binary Numbers Systems



**Range of *N*-bit numbers**

| System | Range |
|---|---|
| Unsigned | $[0, 2^N - 1]$ |
| Sign/Magnitude | $[-2^{N-1} + 1, 2^{N-1} - 1]$ |
| Two’s Complement | $[-2^{N-1}, 2^{N-1} - 1]$ |

# Review: Binary Numbers Systems

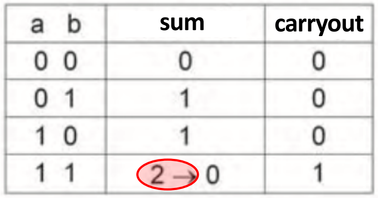


*Number Line*

| | –8 | –7 | –6 | –5 | –4 | –3 | –2 | –1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Unsigned | | | | | | | | | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 |
| Two's Complement | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | | | | | | | | |
| Sign/Magnitude | | 1111 | 1110 | 1101 | 1100 | 1011 | 1010 | 1001 | 0000<br>1000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | | | | | | | | |

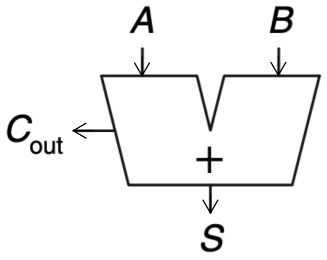
**Number line and 4-bit binary encodings**

# One-bit half adder



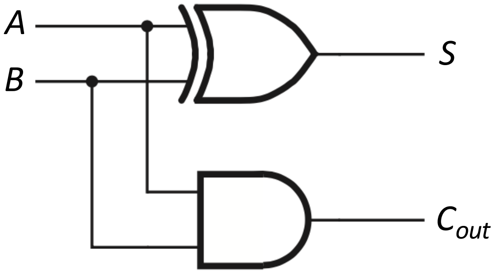
| a b | sum | carryout |
|---|---|---|
| 0 0 | 0 | 0 |
| 0 1 | 1 | 0 |
| 1 0 | 1 | 0 |
| 1 1 | 2 → 0 | 1 |

**Half Adder**



| $A$ | $B$ | $C_{out}$ | $S$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

$$S = A \oplus B$$
$$C_{out} = AB$$

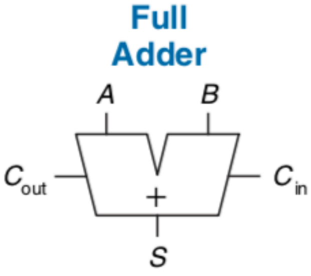


# One-bit Full Adder

```
    1
 0001
+0101
 0110
```

**Figure 5.2** **Carry bit**

*If we need to add multiple bits we must consider whether we have a carry in or not*



| $C_{in}$ | $A$ | $B$ | $C_{out}$ | $S$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

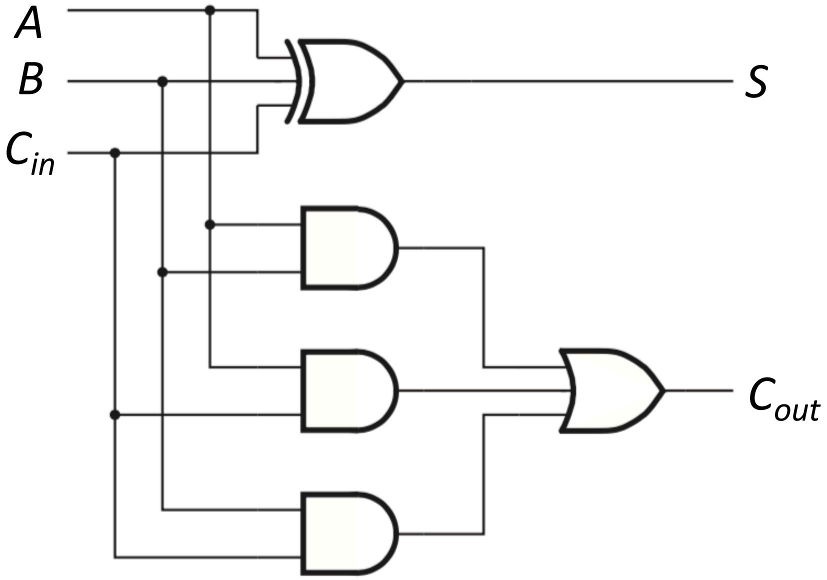
$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AB + AC_{in} + BC_{in}$$

**Figure 5.3** **1-bit full adder**

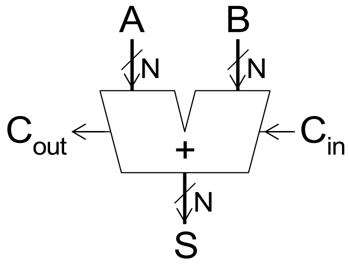
S is high when the number of inputs that are high is odd *(odd-parity function)*

$C_{out}$ is high when two or more inputs are high (*majority function*)

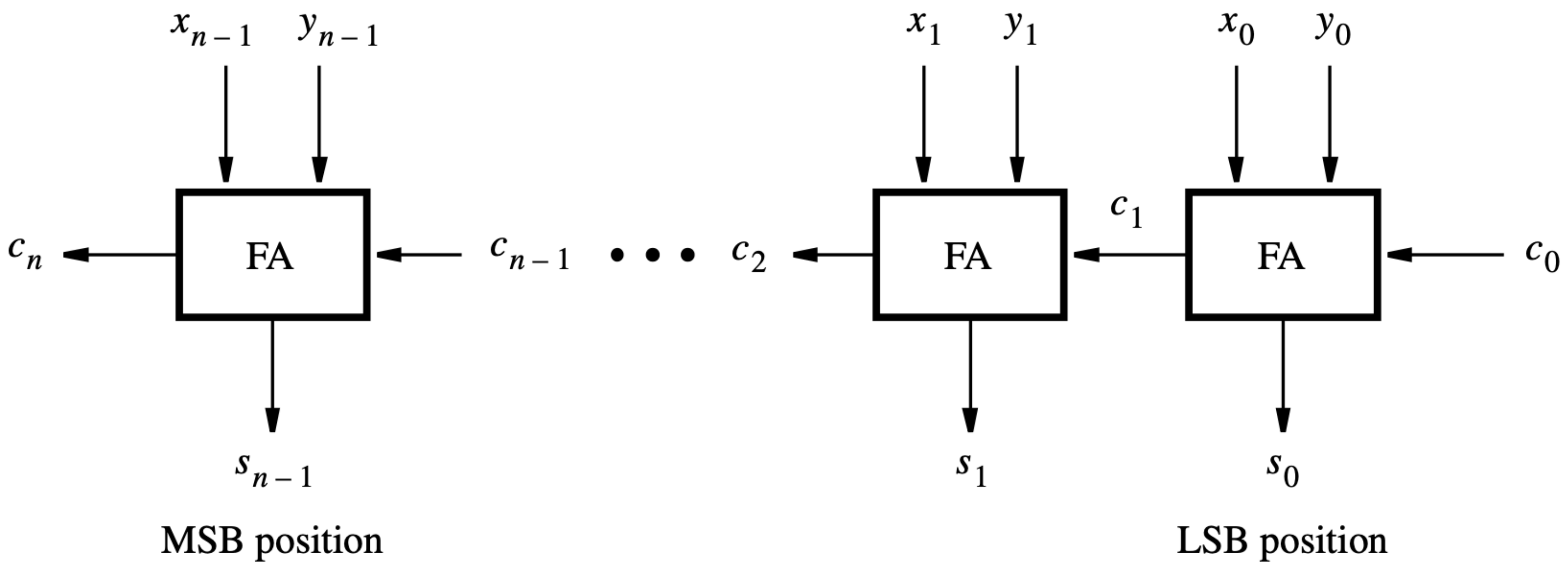


# Multi-bit adder (carry-ripple)

**Symbol for multi-bit adder**

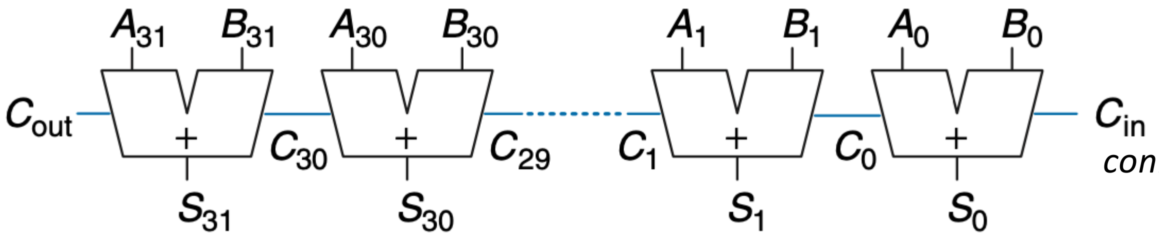


One possible multi-bit adder implementation is given by the carry-ripple architecture



*In the LSB position we can use a Half Adder or a Full Adder with $c_0$ connected to GND*

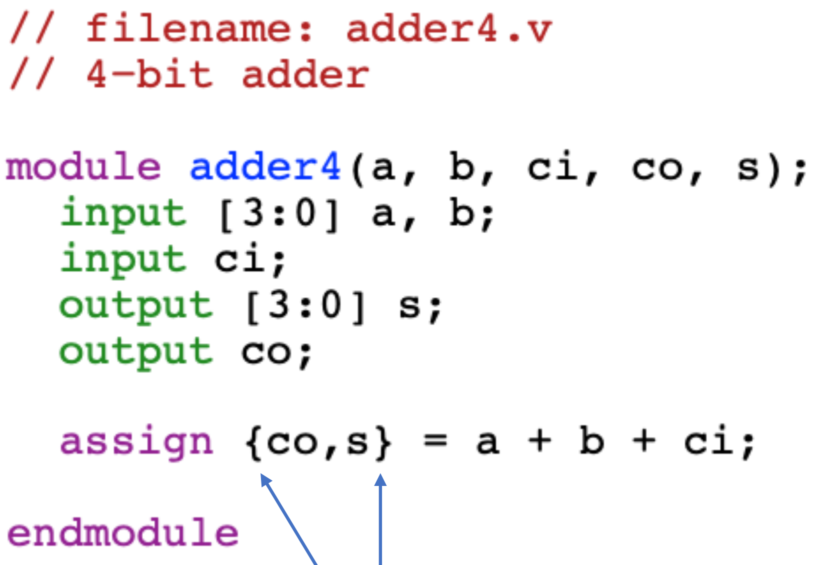
# Example: 32-bit ripple-carry adder



*connect $C_{in}$ to ground*

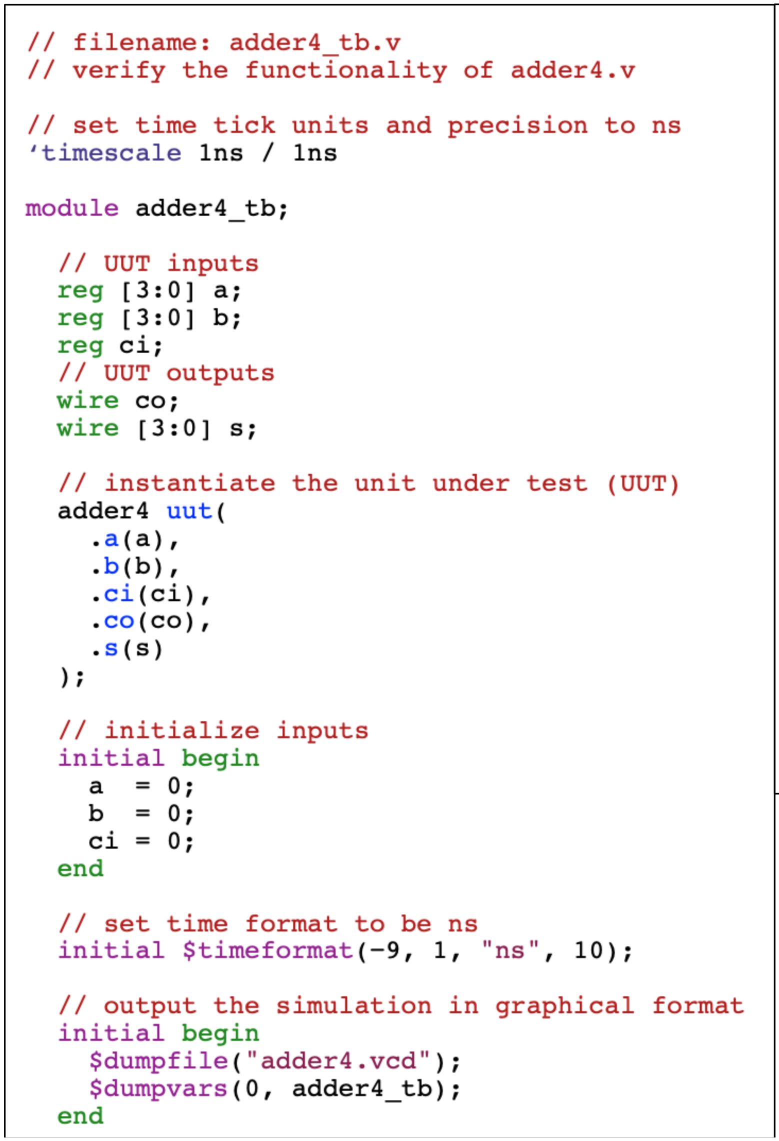
**Figure 5.5** **32-bit ripple-carry adder**

# Example: 4-bit adder in Verilog

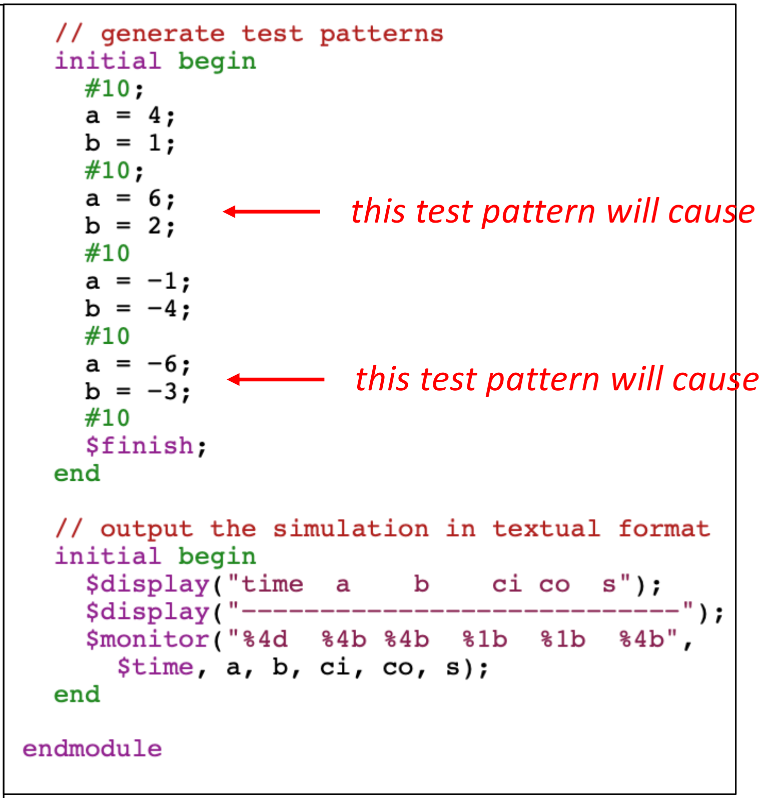


```
// filename: adder4.v
// 4-bit adder

module adder4(a, b, ci, co, s);
  input [3:0] a, b;
  input ci;
  output [3:0] s;
  output co;

  assign {co,s} = a + b + ci;

endmodule
```

*concatenation*



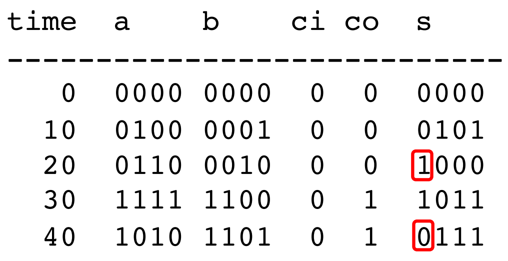
```
// filename: adder4_tb.v
// verify the functionality of adder4.v

// set time tick units and precision to ns
‘timescale 1ns / 1ns

module adder4_tb;

  // UUT inputs
  reg [3:0] a;
  reg [3:0] b;
  reg ci;
  // UUT outputs
  wire co;
  wire [3:0] s;

  // instantiate the unit under test (UUT)
  adder4 uut(
    .a(a),
    .b(b),
    .ci(ci),
    .co(co),
    .s(s)
  );

  // initialize inputs
  initial begin
    a  = 0;
    b  = 0;
    ci = 0;
  end

  // set time format to be ns
  initial $timeformat(-9, 1, "ns", 10);

  // output the simulation in graphical format
  initial begin
    $dumpfile("adder4.vcd");
    $dumpvars(0, adder4_tb);
  end
```



```
  // generate test patterns
  initial begin
    #10;
    a = 4;
    b = 1;
    #10;
    a = 6;
    b = 2;
    #10
    a = -1;
    b = -4;
    #10
    a = -6;
    b = -3;
    #10
    $finish;
  end

  // output the simulation in textual format
  initial begin
    $display("time  a    b    ci co  s");
    $display("----------------------------");
    $monitor("%4d  %4b %4b  %1b  %1b  %4b",
      $time, a, b, ci, co, s);
  end

endmodule
```

*this test pattern will cause overflow* (a = 6; b = 2;)

*this test pattern will cause overflow* (a = -6; b = -3;)

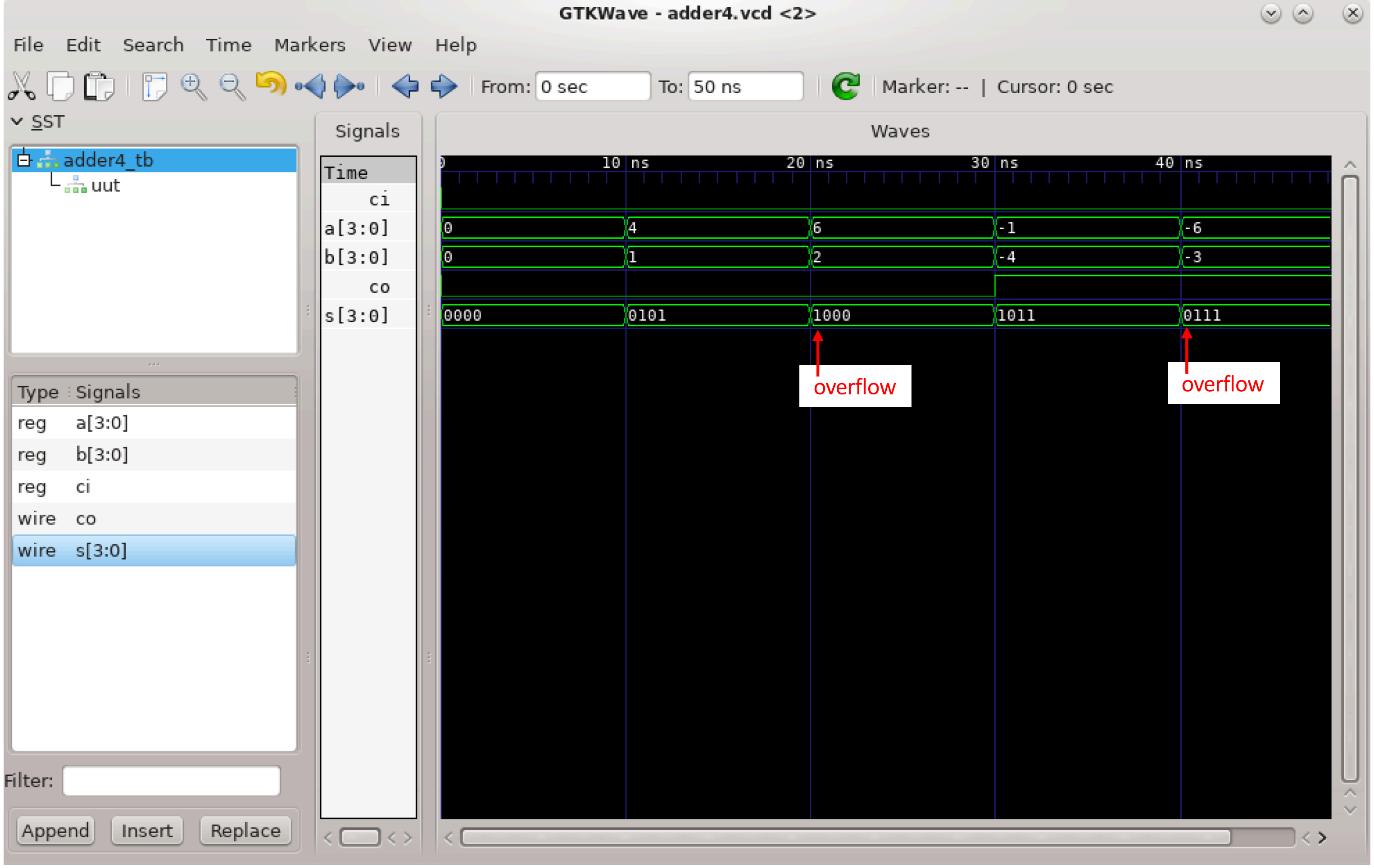


```
time  a    b    ci co  s
----------------------------
   0  0000 0000  0  0  0000
  10  0100 0001  0  0  0101
  20  0110 0010  0  0  1000
  30  1111 1100  0  1  1011
  40  1010 1101  0  1  0111
```

overflow (20)

overflow (40)

# Example: 4-bit adder in Verilog



GTKWave - adder4.vcd <2>

From: 0 sec To: 50 ns

Marker: -- | Cursor: 0 sec

| Type | Signals |
|---|---|
| reg | a[3:0] |
| reg | b[3:0] |
| reg | ci |
| wire | co |
| wire | s[3:0] |

overflow

overflow

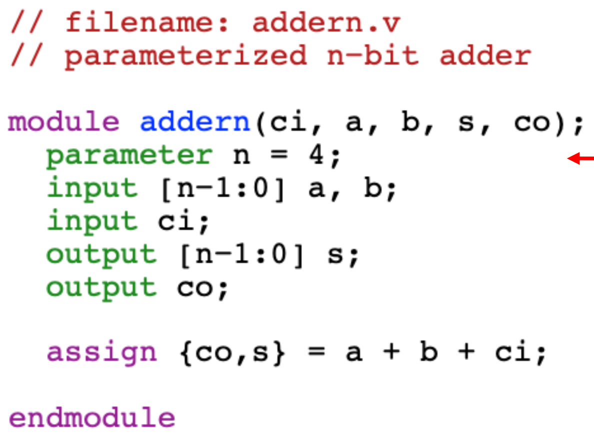
# Overflow (N bit adder)

$$V = (a[N-1]\ \&\ b[N-1]\ \&\ {\sim}s[N-1]) | ({\sim}a[N-1]\ \&\ {\sim}b[N-1]\ \&\ s[N-1])$$

*a = operand 1* *b = operand 2* *s = sum*

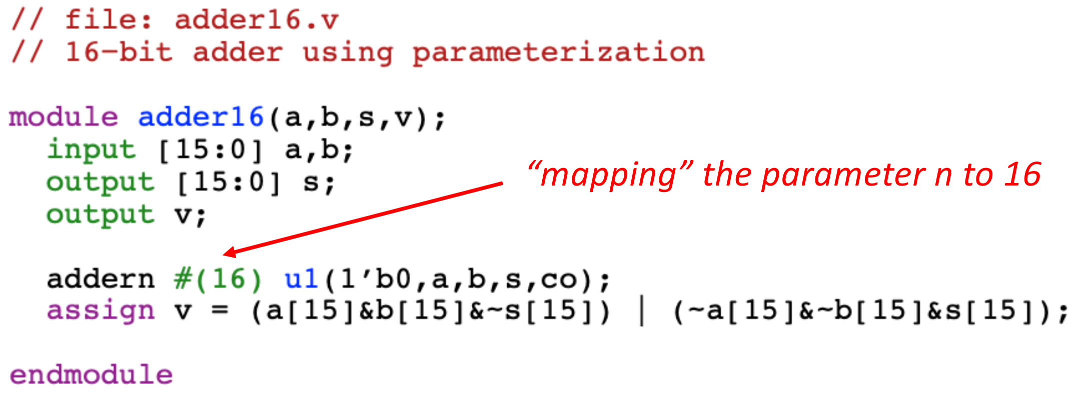
$$V = s[N-1] \oplus cout$$

# Example: 16 bit adder in Verilog



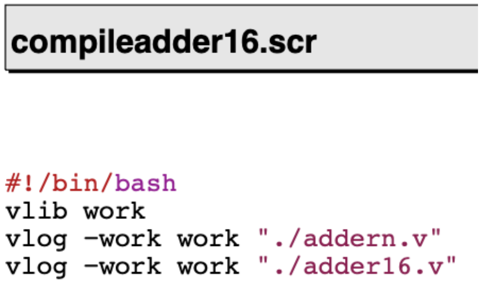
```
// filename: addern.v
// parameterized n-bit adder

module addern(ci, a, b, s, co);
  parameter n = 4;
  input [n-1:0] a, b;
  input ci;
  output [n-1:0] s;
  output co;

  assign {co,s} = a + b + ci;

endmodule
```

*defining a parameter n with default value 4*



```
// file: adder16.v
// 16-bit adder using parameterization

module adder16(a,b,s,v);
  input [15:0] a,b;
  output [15:0] s;
  output v;

  addern #(16) u1(1’b0,a,b,s,co);
  assign v = (a[15]&b[15]&~s[15]) | (~a[15]&~b[15]&s[15]);

endmodule
```

*“mapping” the parameter n to 16*

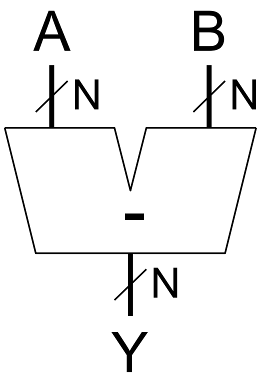


**compileadder16.scr**

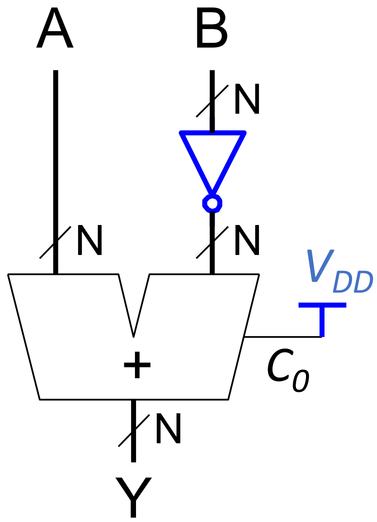
```
#!/bin/bash
vlib work
vlog -work work "./addern.v"
vlog -work work "./adder16.v"
```

# Multi-bit subtractor

**Symbol**

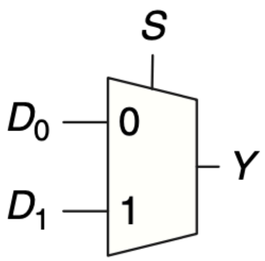


**Implementation**



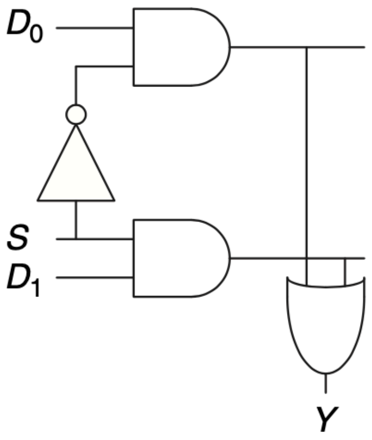
*Got a new appreciation for complement-2 notation ?*

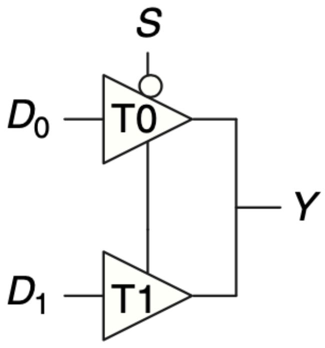
# 2:1 Mux



| $S$ | $D_1$ | $D_0$ | $Y$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

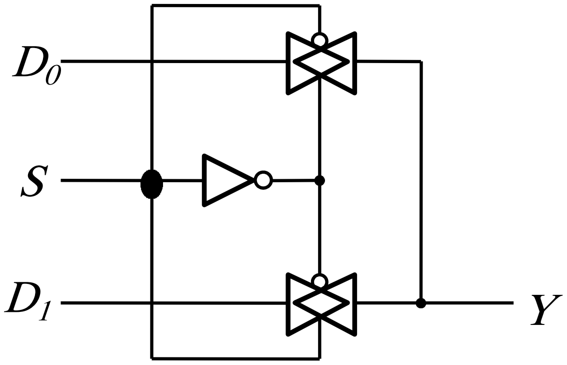
**Figure 2.54** **2:1 multiplexer symbol and truth table**





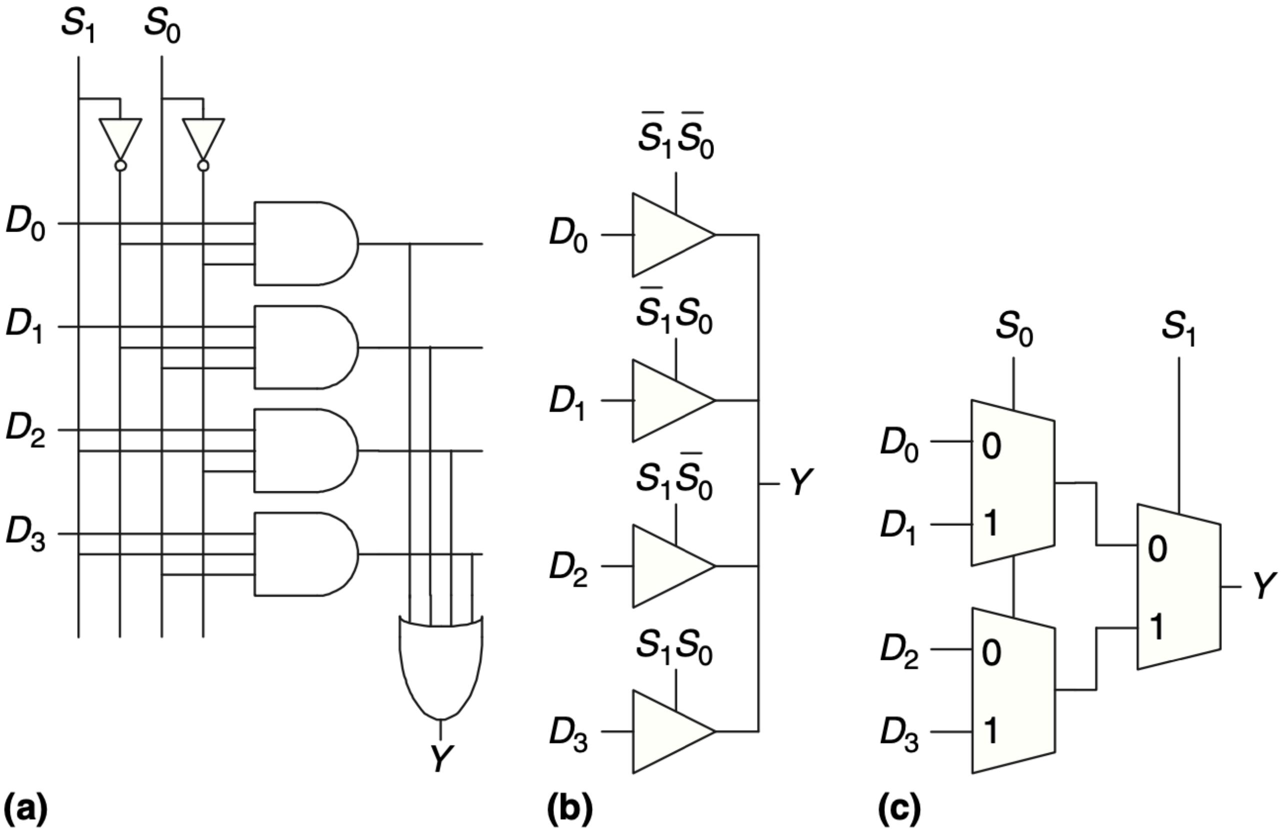
$$Y = D_0\overline{S} + D_1 S$$

**Figure 2.56** **Multiplexer using tristate buffers**



***Circuit with transmission gates***

# 4:1 Mux



**Various implementation of 4:1 multiplexer**

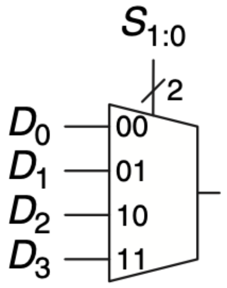
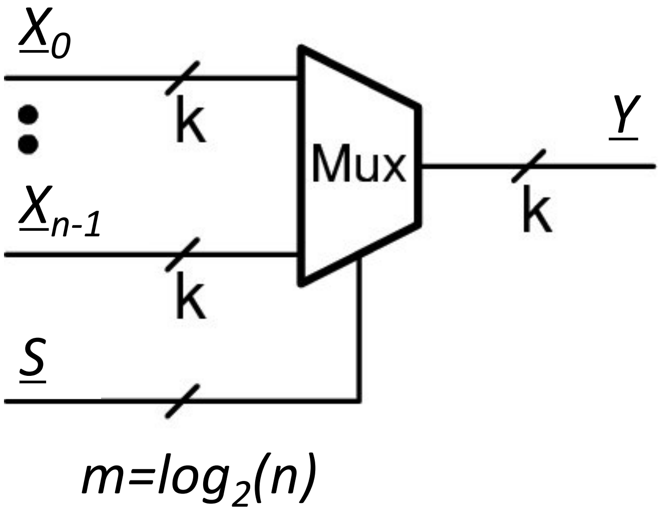


**Figure 2.57** **4:1 multiplexer**

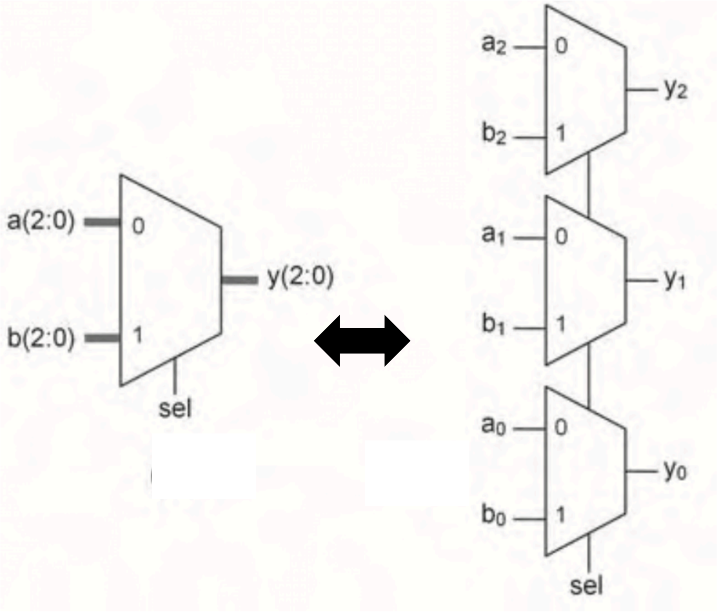
# Mux with multi-bit inputs

*A multiplexer is a data selector*



*m=log$_2$(n)*

Example



# Verilog Example (32 bit wide 4-inputs mux)

```
// filename: mux41_32b.v
// 32 bit wide 4:1 mux

module mux41_32b(s,d3, d2, d1, d0, y);
  input  [1:0] s;
  input  [31:0] d3, d2, d1, d0;
  output [31:0] y;

  reg [31:0] y;

  always @*
  begin
    case (s)
      2'b00: y = d0;
      2'b01: y = d1;
      2'b10: y = d2;
      default: y = d3;
    endcase
  end
endmodule
```

*IMPORTANT ASIDE: I "originally" forgot to put [31:0] in front of y and iverilog did not catch it (fortunately Modelsim caught it!)*

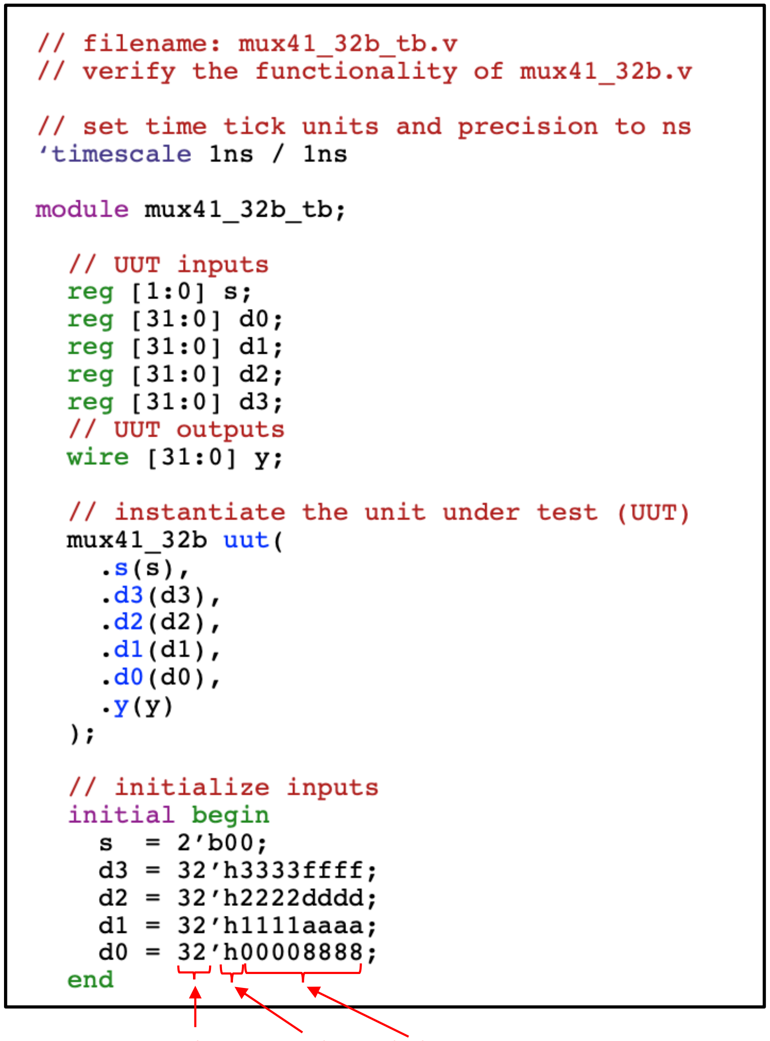
**runmux41.scr**

```
#!/bin/bash
vlib work
vlog -work work "./mux41_32b.v"
vlog -work work "./mux41_32b_tb.v"
vsim work.mux41_32b_tb -do simmux41.do
```

```
# data:
# -------------
# d0 = 00008888
# d1 = 1111aaaa
# d2 = 2222dddd
# d3 = 3333ffff
#
# time  s y
# ----------------
#    0  0 00008888
#   10  1 1111aaaa
#   20  2 2222dddd
#   30  3 3333ffff
```

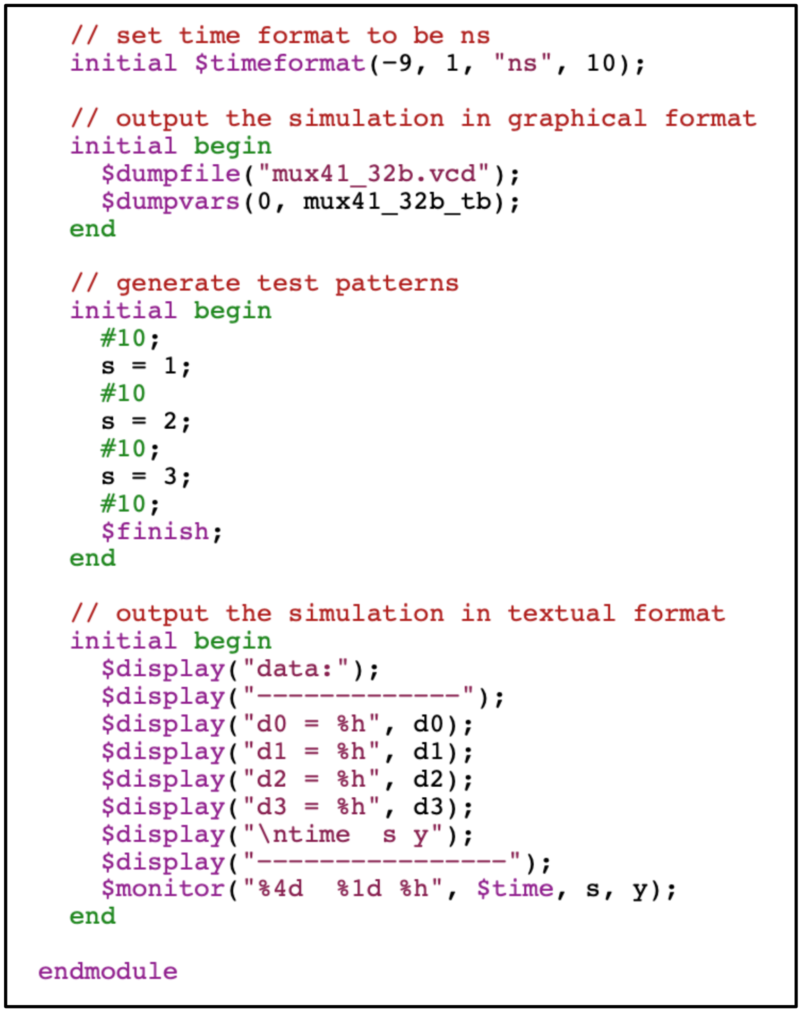
**simmux41.do**

```
restart -f
add wave -radix hex sim:/mux41_32b_tb/*
run -all
# quit
```



```verilog
// filename: mux41_32b_tb.v
// verify the functionality of mux41_32b.v

// set time tick units and precision to ns
‘timescale 1ns / 1ns

module mux41_32b_tb;

  // UUT inputs
  reg [1:0] s;
  reg [31:0] d0;
  reg [31:0] d1;
  reg [31:0] d2;
  reg [31:0] d3;
  // UUT outputs
  wire [31:0] y;

  // instantiate the unit under test (UUT)
  mux41_32b uut(
    .s(s),
    .d3(d3),
    .d2(d2),
    .d1(d1),
    .d0(d0),
    .y(y)
  );

  // initialize inputs
  initial begin
    s  = 2’b00;
    d3 = 32’h3333ffff;
    d2 = 32’h2222dddd;
    d1 = 32’h1111aaaa;
    d0 = 32’h00008888;
  end
```

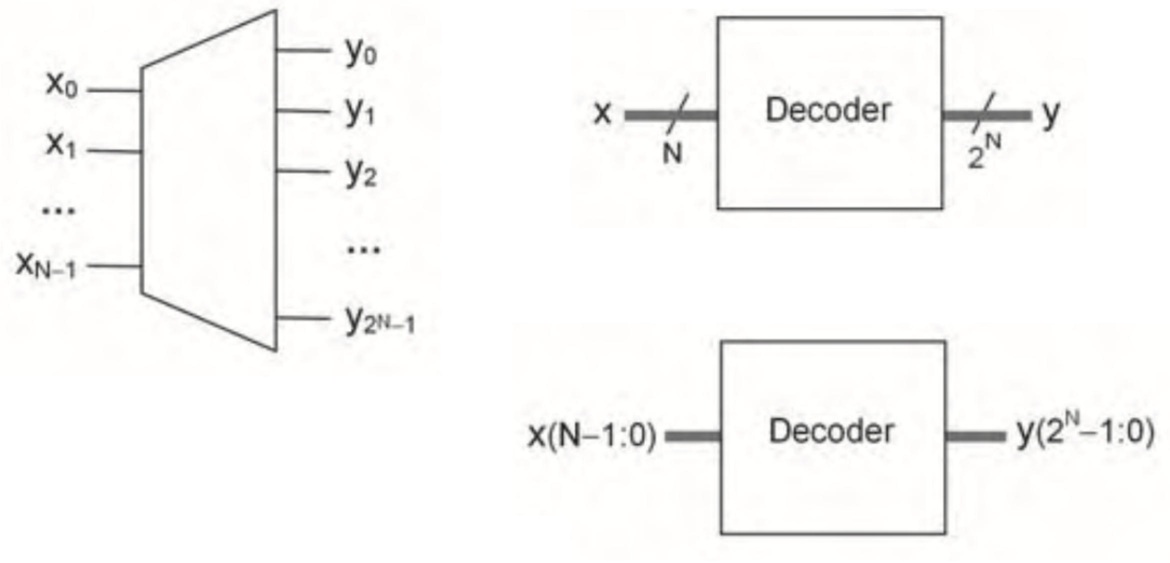
<number of bits>’<radix><digits>



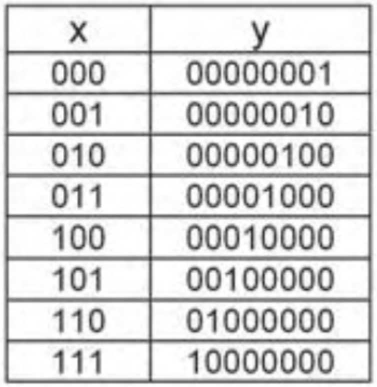
```verilog
  // set time format to be ns
  initial $timeformat(-9, 1, "ns", 10);

  // output the simulation in graphical format
  initial begin
    $dumpfile("mux41_32b.vcd");
    $dumpvars(0, mux41_32b_tb);
  end

  // generate test patterns
  initial begin
    #10;
    s = 1;
    #10
    s = 2;
    #10;
    s = 3;
    #10;
    $finish;
  end

  // output the simulation in textual format
  initial begin
    $display("data:");
    $display("-------------");
    $display("d0 = %h", d0);
    $display("d1 = %h", d1);
    $display("d2 = %h", d2);
    $display("d3 = %h", d3);
    $display("\ntime  s y");
    $display("----------------");
    $monitor("%4d  %1d %h", $time, s, y);
  end

endmodule
```

# Decoder

- Converts an N-bit input into a $2^N$-bit output with the output having only one “hot-bit”



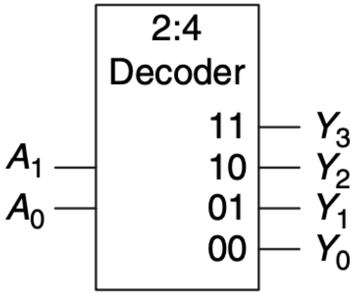
*Decoder symbols*



| x | y |
|---|---|
| 000 | 00000001 |
| 001 | 00000010 |
| 010 | 00000100 |
| 011 | 00001000 |
| 100 | 00010000 |
| 101 | 00100000 |
| 110 | 01000000 |
| 111 | 10000000 |

*Truth Table for N=3*
*(3-bit decoder)*

# Example. 2:4 Decoder



| $A_1$ | $A_0$ | $Y_3$ | $Y_2$ | $Y_1$ | $Y_0$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 |

**Figure 2.63** **2:4 decoder**

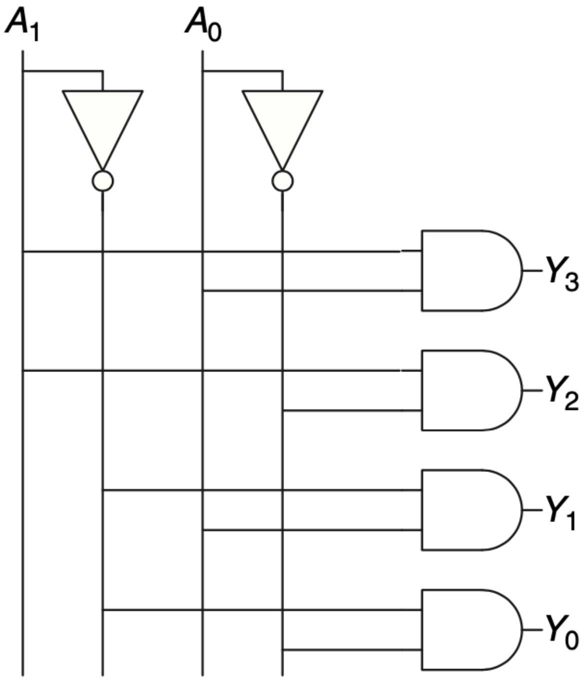
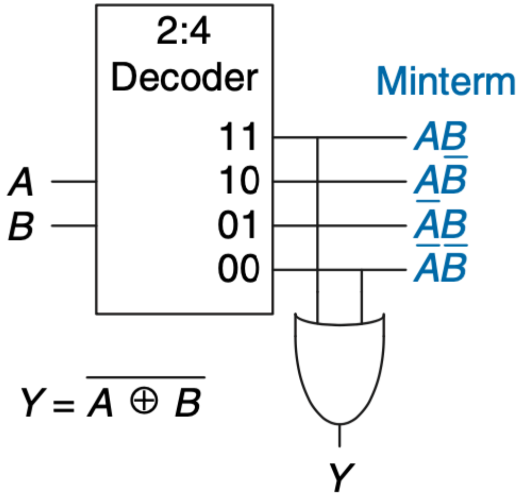


**Figure 2.64** **2:4 decoder implementation**

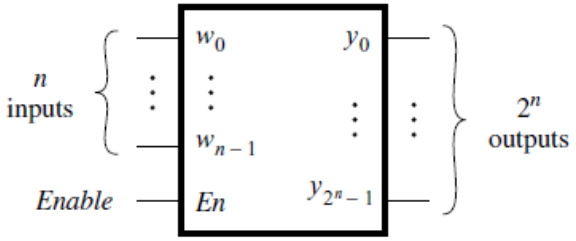
Decoders can be combined with OR gates to build logic functions.



$Y = \overline{A \oplus B}$

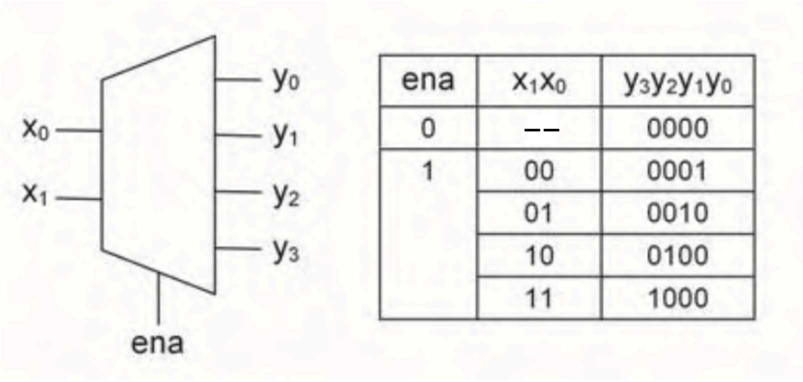
**Figure 2.65** **Logic function using decoder**

# Decoder with Enable

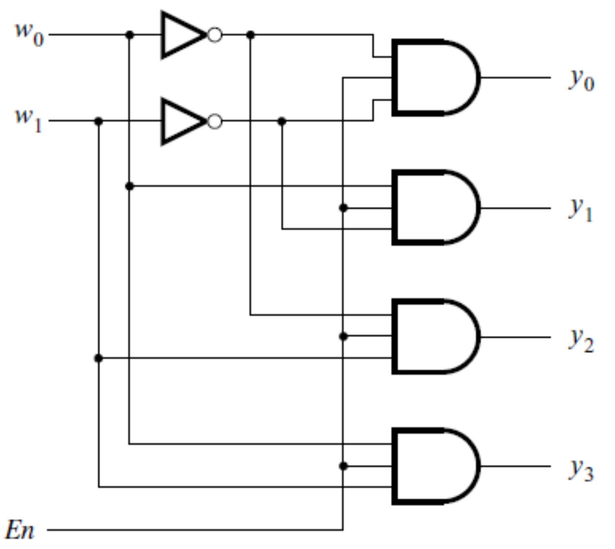


An n-to-$2^n$ decoder

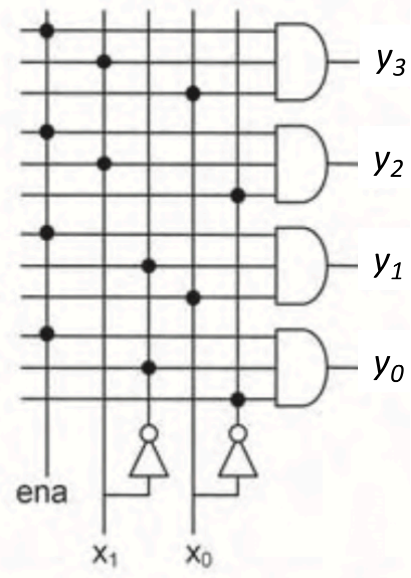
*Example. 2 to 4 decoder with enable*



| ena | $x_1x_0$ | $y_3y_2y_1y_0$ |
|---|---|---|
| 0 | -- | 0000 |
| 1 | 00 | 0001 |
|  | 01 | 0010 |
|  | 10 | 0100 |
|  | 11 | 1000 |

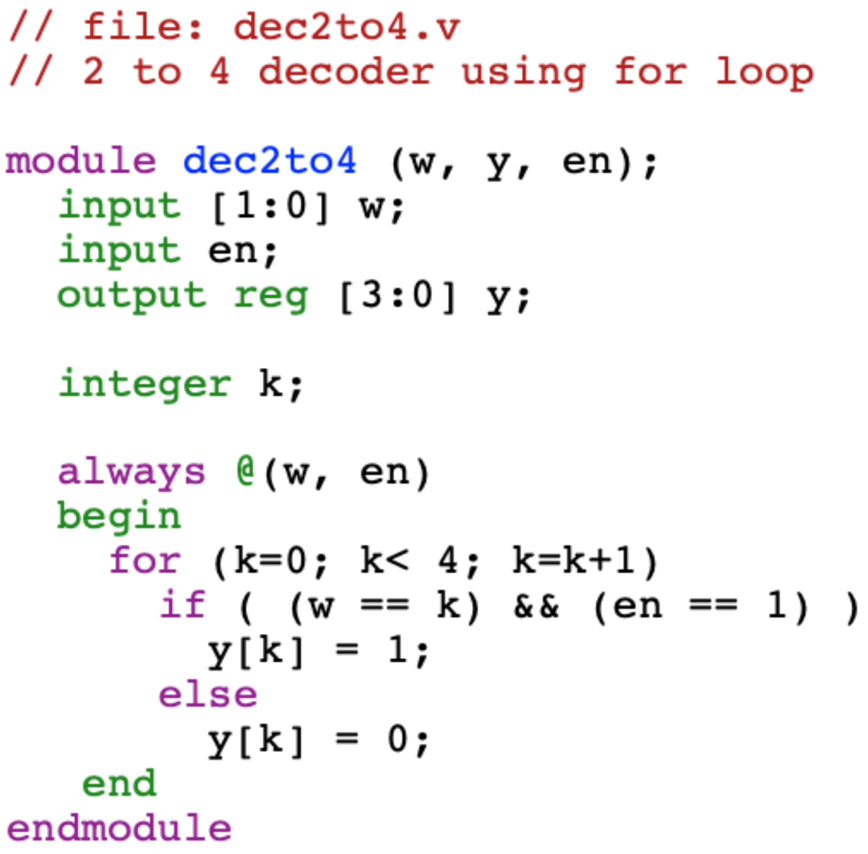


Logic circuit



Logic circuit
(more compact notation)

# Verilog example: 2 to 4 decoder with enable

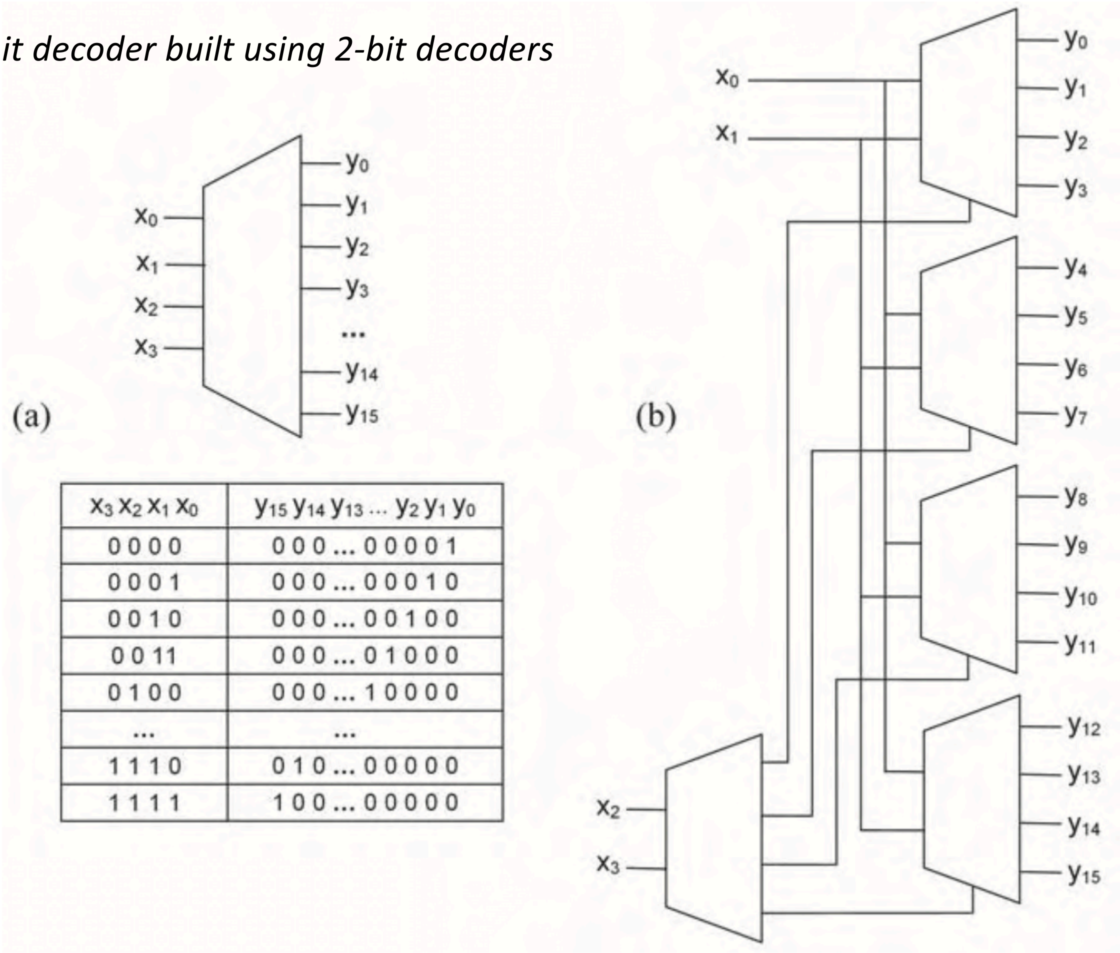


```
// file: dec2to4.v
// 2 to 4 decoder using for loop

module dec2to4 (w, y, en);
  input [1:0] w;
  input en;
  output reg [3:0] y;

  integer k;

  always @(w, en)
  begin
    for (k=0; k< 4; k=k+1)
      if ( (w == k) && (en == 1) )
        y[k] = 1;
      else
        y[k] = 0;
   end
endmodule
```

```
time  en w   y
-----------------
   0  0  00  0000
  10  0  01  0000
  20  0  10  0000
  30  0  11  0000
  40  1  00  0001
  50  1  01  0010
  60  1  10  0100
  70  1  11  1000
```

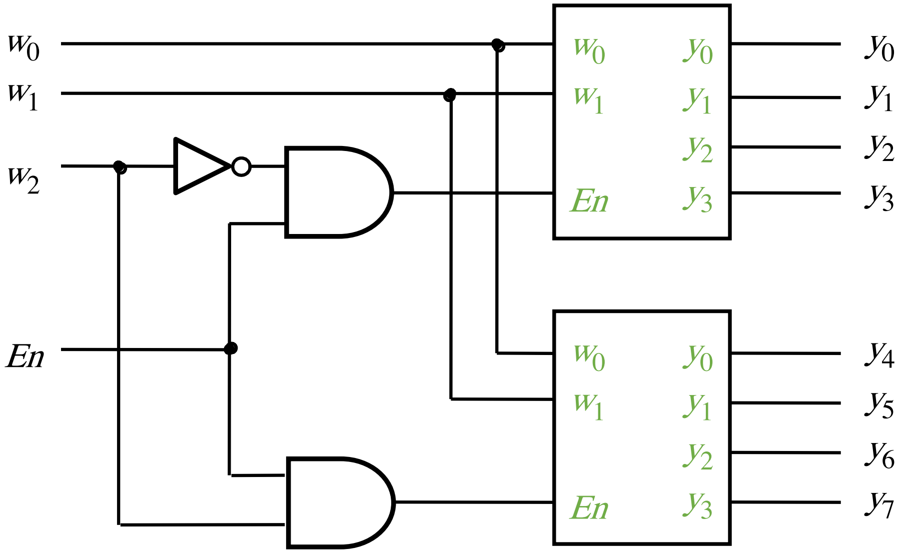
# Large Decoder

*Example.* *4-bit decoder built using 2-bit decoders*



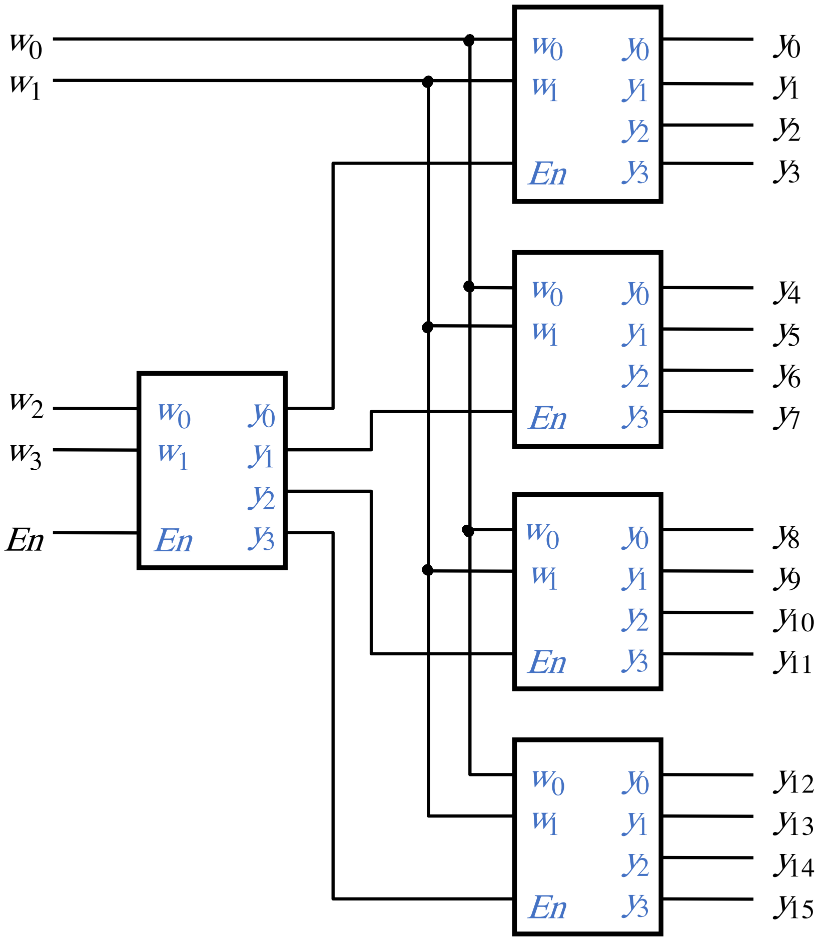
| $x_3\, x_2\, x_1\, x_0$ | $y_{15}\, y_{14}\, y_{13} \ldots y_2\, y_1\, y_0$ |
|---|---|
| 0 0 0 0 | 0 0 0 ... 0 0 0 0 1 |
| 0 0 0 1 | 0 0 0 ... 0 0 0 1 0 |
| 0 0 1 0 | 0 0 0 ... 0 0 1 0 0 |
| 0 0 1 1 | 0 0 0 ... 0 1 0 0 0 |
| 0 1 0 0 | 0 0 0 ... 1 0 0 0 0 |
| ... | ... |
| 1 1 1 0 | 0 1 0 ... 0 0 0 0 0 |
| 1 1 1 1 | 1 0 0 ... 0 0 0 0 0 |

# Example



3-to-8 decoder with enable using two 2-to-4 decoders.

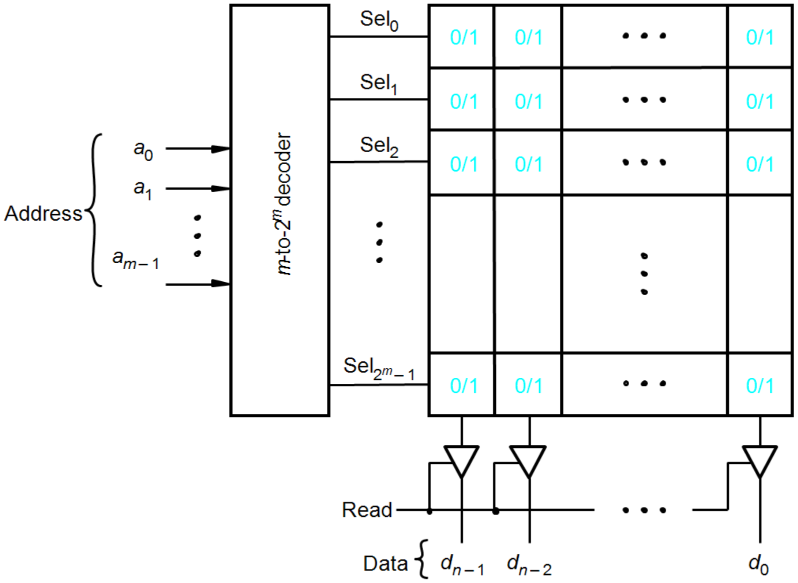
# Example



4-to-16 decoder with enable
built using a decoder tree.

# Decoder Application

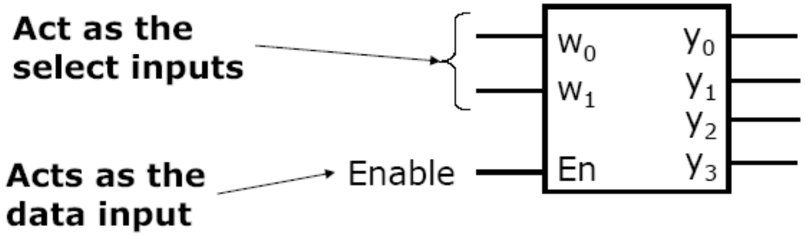
- A common decoder application is decoding the address lines of memory chips



$2^m$ x $n$-bit read-only memory (ROM) block.

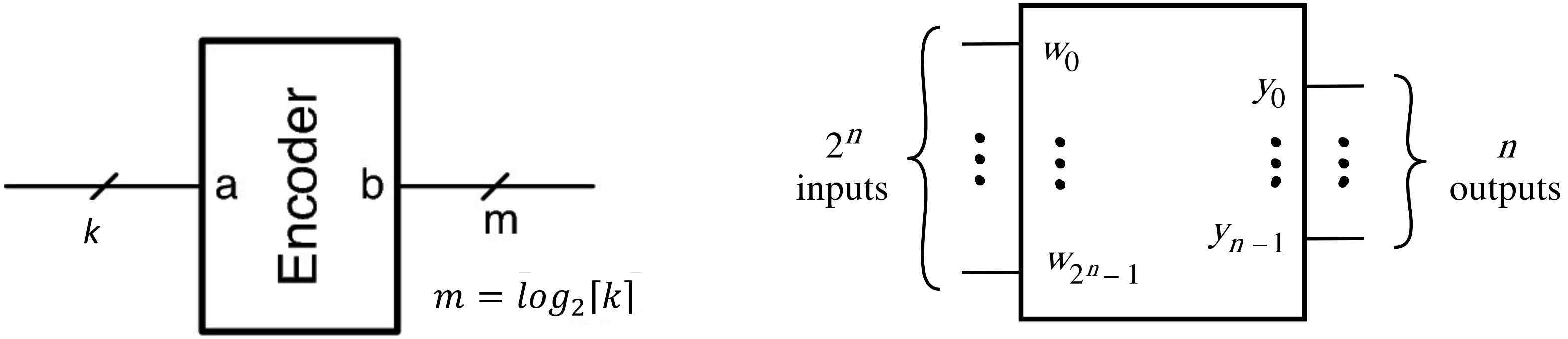
# Another Decoder application: Demultiplexers

- A demultiplexer places a single data input onto one of the multiple outputs
- A *1*-to-2*ⁿ* demultiplexer can be implemented using an *n*-to-2*ⁿ* decoder



# Encoder

- An encoder converts its $2^N$ inputs (of which only one is hot) into N outputs.

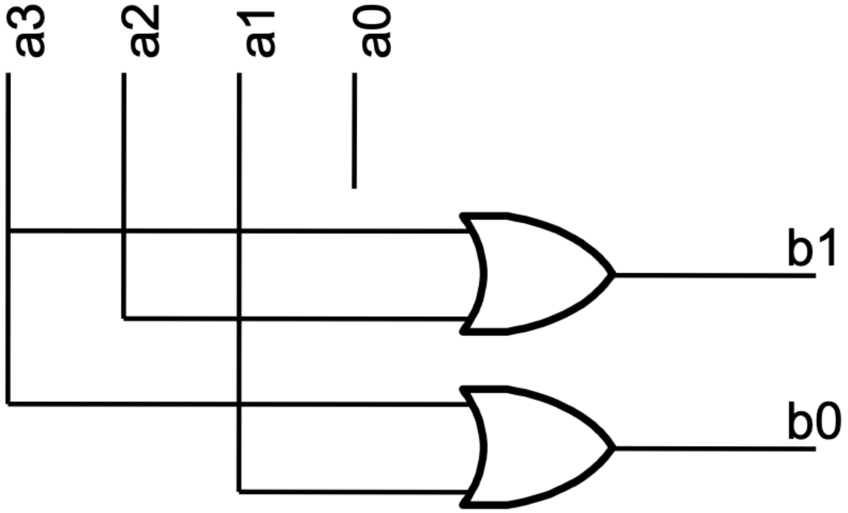


*Symbols*

# Example. 4 to 2 Encoder

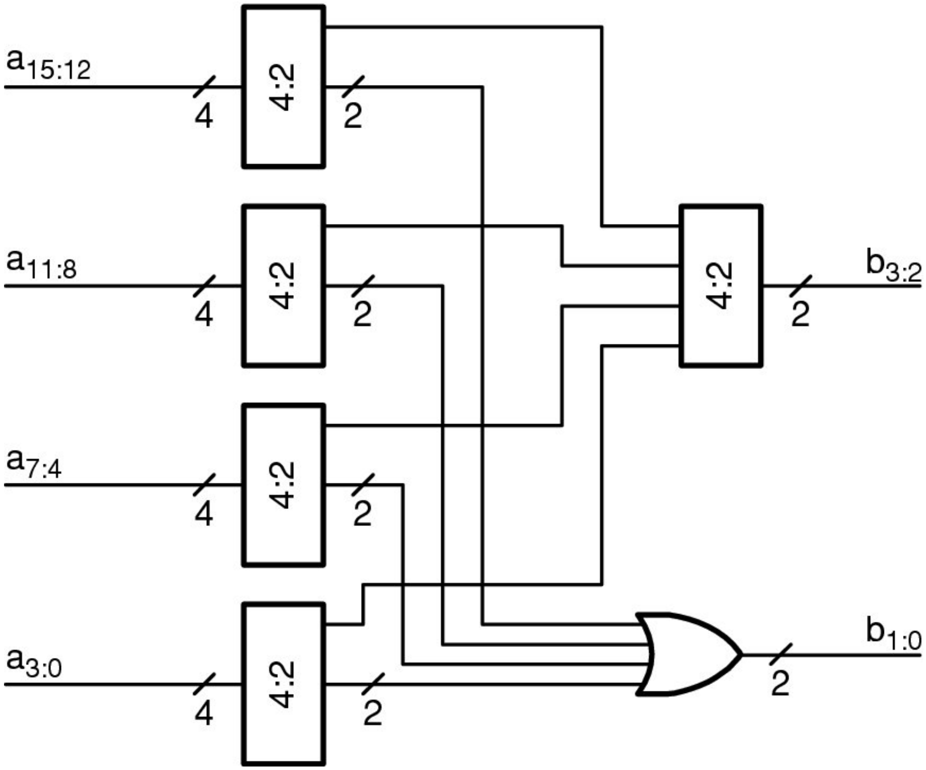
| a3 | a2 | a1 | a0 | b1 | b0 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 |

*b0 = a3 + a1*
*b1 = a3 + a2*



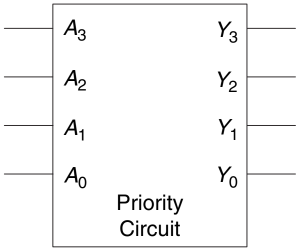
# Large Encoders

- Large encoders can be built from a tree of smaller encoders

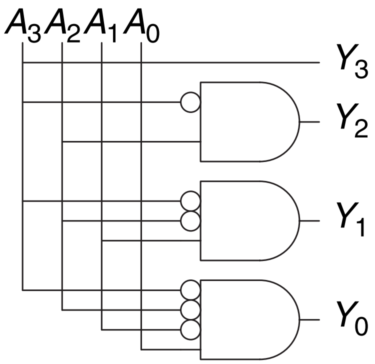


*Example. 16 to 4 encoder*

# Priority Encoder



| $A_3$ | $A_2$ | $A_1$ | $A_0$ | $Y_3$ | $Y_2$ | $Y_1$ | $Y_0$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |

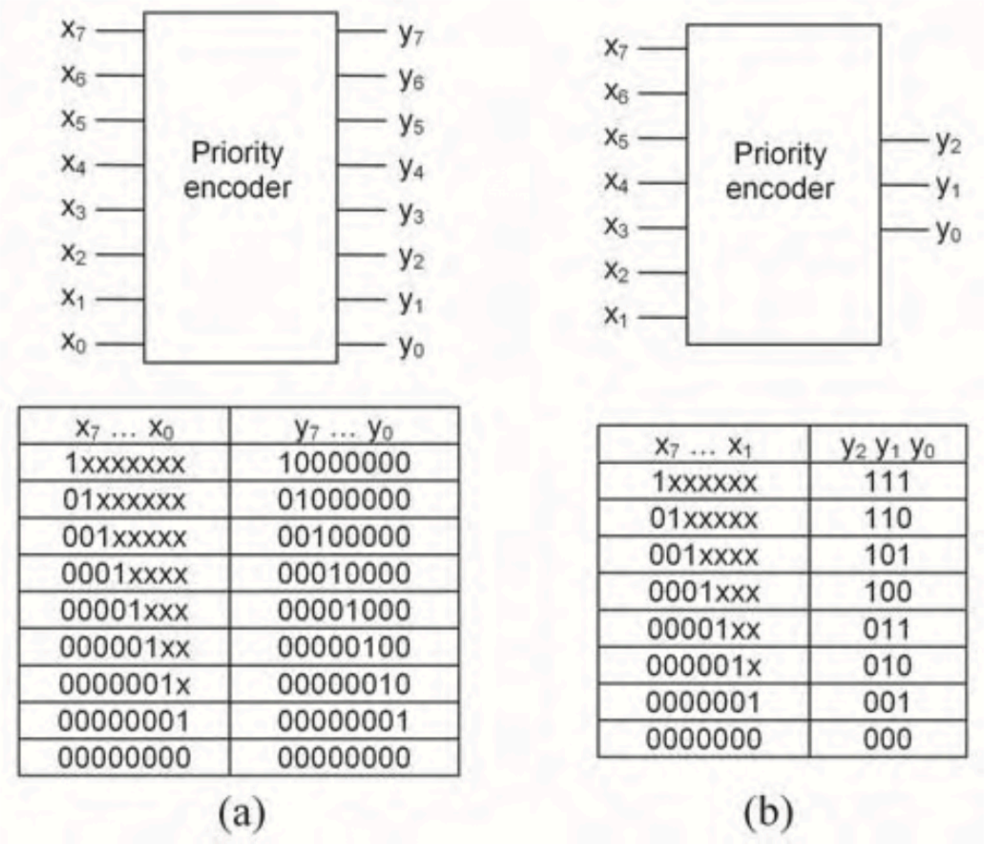


| $A_3$ | $A_2$ | $A_1$ | $A_0$ | $Y_3$ | $Y_2$ | $Y_1$ | $Y_0$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | X | 0 | 0 | 1 | 0 |
| 0 | 1 | X | X | 0 | 1 | 0 | 0 |
| 1 | X | X | X | 1 | 0 | 0 | 0 |

A more concise truth table ( with don’t cares as X)

# Priority Encoders

- Priority encoders can come in two flavors



| $x_7 \ldots x_0$ | $y_7 \ldots y_0$ |
|---|---|
| 1xxxxxxx | 10000000 |
| 01xxxxxx | 01000000 |
| 001xxxxx | 00100000 |
| 0001xxxx | 00010000 |
| 00001xxx | 00001000 |
| 000001xx | 00000100 |
| 0000001x | 00000010 |
| 00000001 | 00000001 |
| 00000000 | 00000000 |

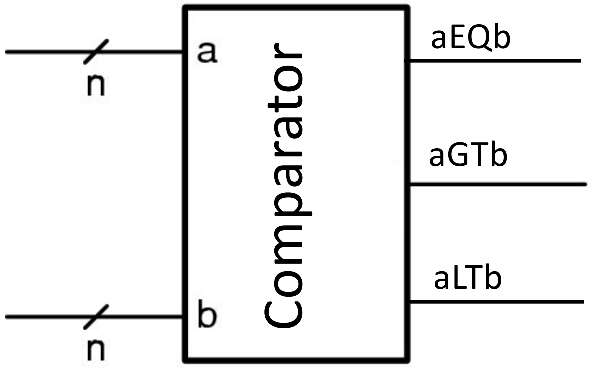
(a)

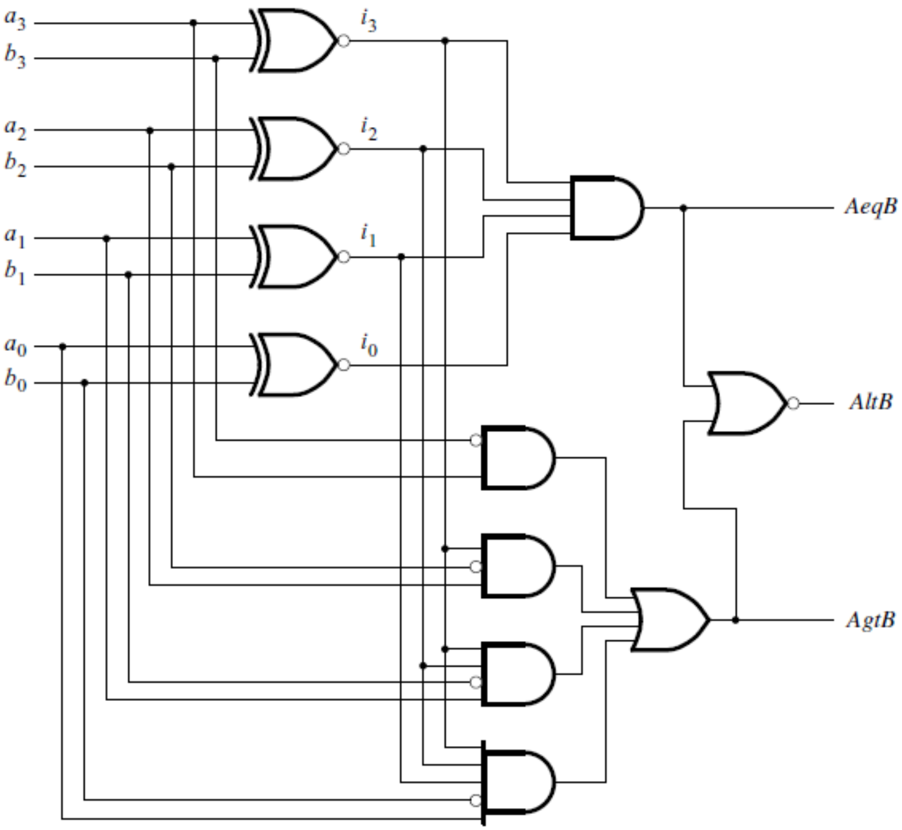
| $x_7 \ldots x_1$ | $y_2\, y_1\, y_0$ |
|---|---|
| 1xxxxxx | 111 |
| 01xxxxx | 110 |
| 001xxxx | 101 |
| 0001xxx | 100 |
| 00001xx | 011 |
| 000001x | 010 |
| 0000001 | 001 |
| 0000000 | 000 |

(b)

(b) the output displays the address of the input bit with highest priority

# Comparator

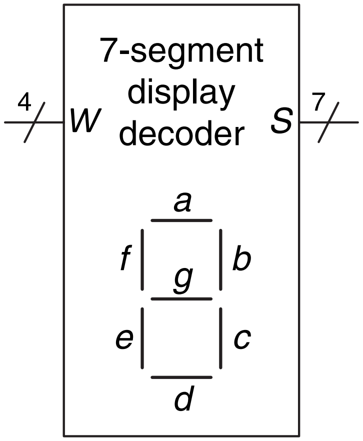


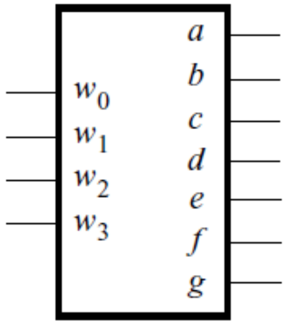


*Example: A four-bit comparator circuit.*

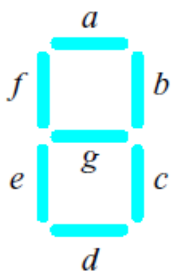
# Code Converters

- Example: HEX to 7-seg. Display





(a) Code converter

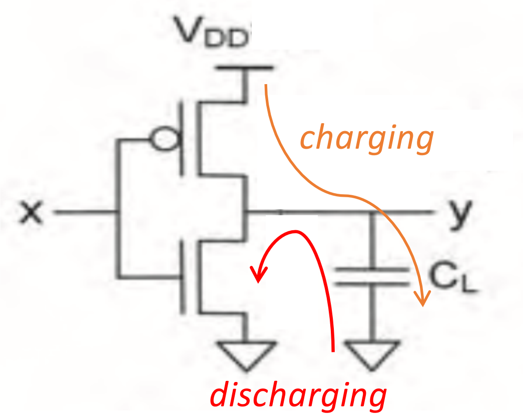


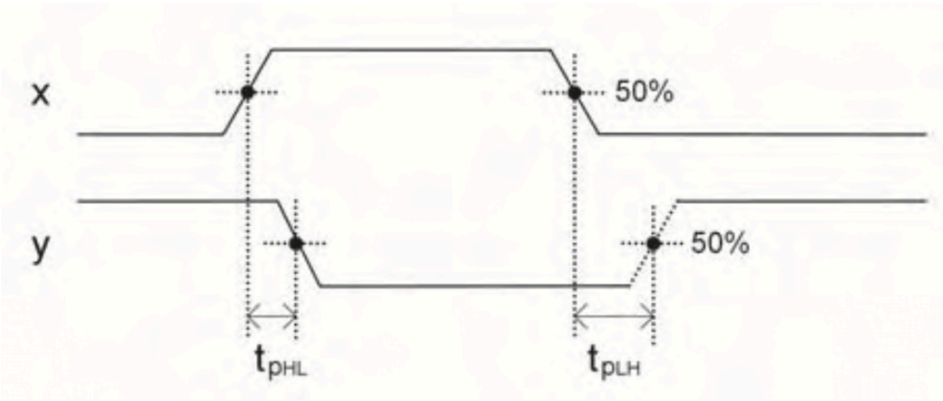
(b) 7-segment display

| $w_3$ | $w_2$ | $w_1$ | $w_0$ | $a$ | $b$ | $c$ | $d$ | $e$ | $f$ | $g$ |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |

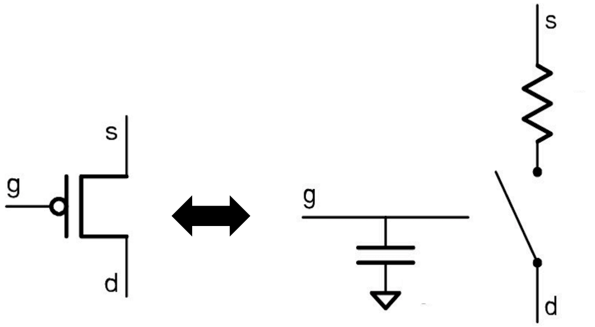
(c) Truth table

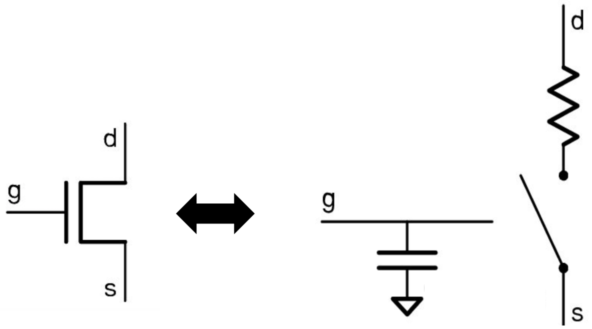
# Delays



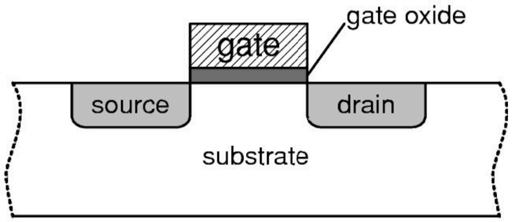


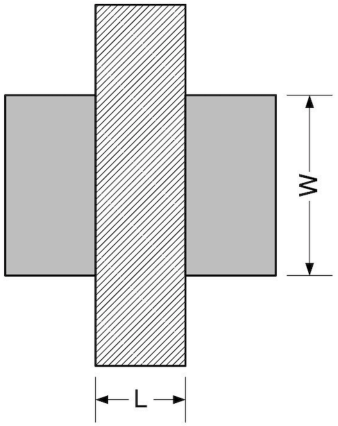
- Delay is caused by capacitance and resistance



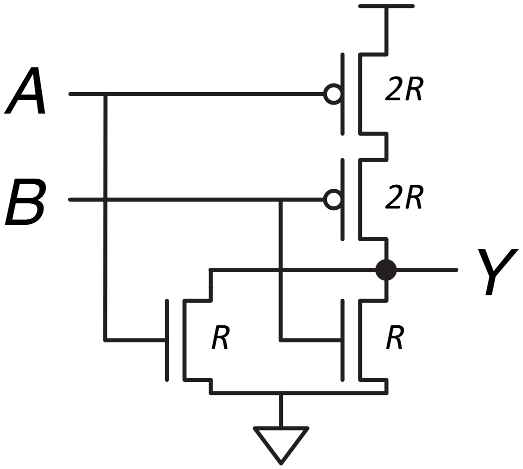


# Delays

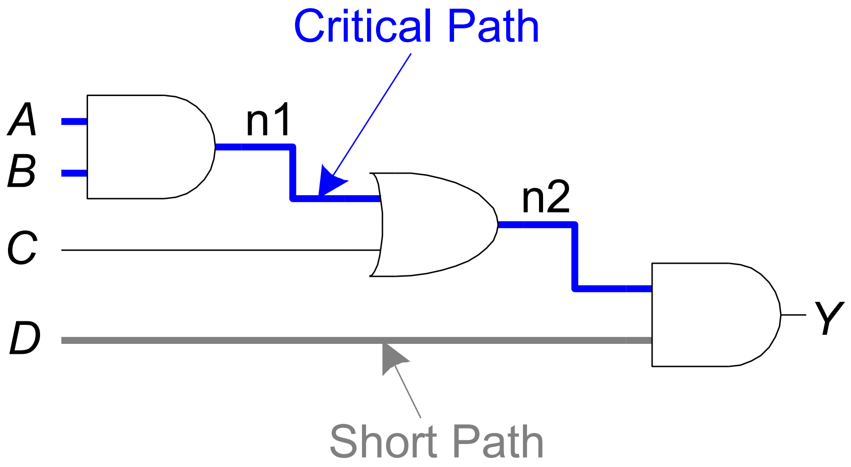




- Propagation delay: $t_{pd}$ = max delay from input to output
- Propagation delay varies with
  - rise and fall time of the input signals
  - outputs switching
    (what outputs switch depends on what inputs switch)
  - temperature, supply voltage, geometry
  - capacitance $C_L$ driven



# Critical (long) and Short Paths

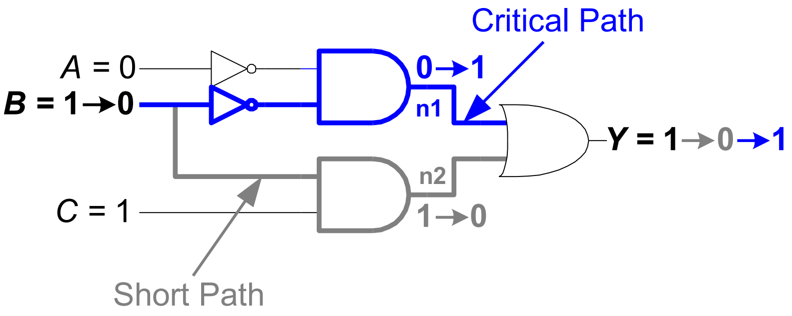


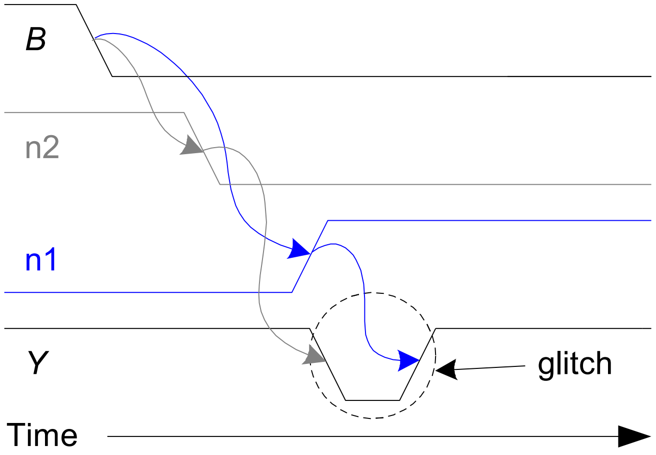
Critical (Long) Path: $t_{pd} = 2t_{pd\_\mathrm{AND}} + t_{pd\_\mathrm{OR}}$

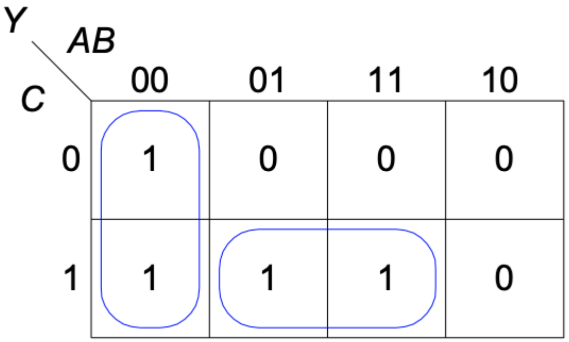
Short Path: $t_{cd} = t_{cd\_\mathrm{AND}}$

# Glitches

- A single input change causes an output to change multiple times





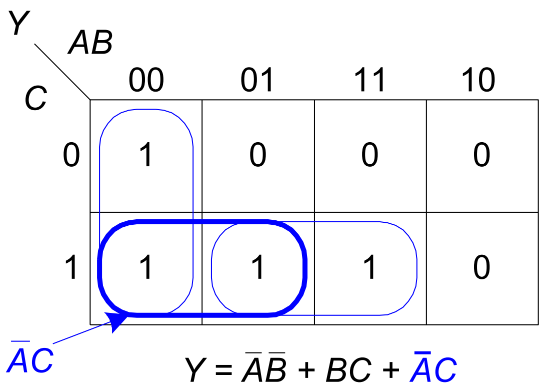


| Y (C \ AB) | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 |

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

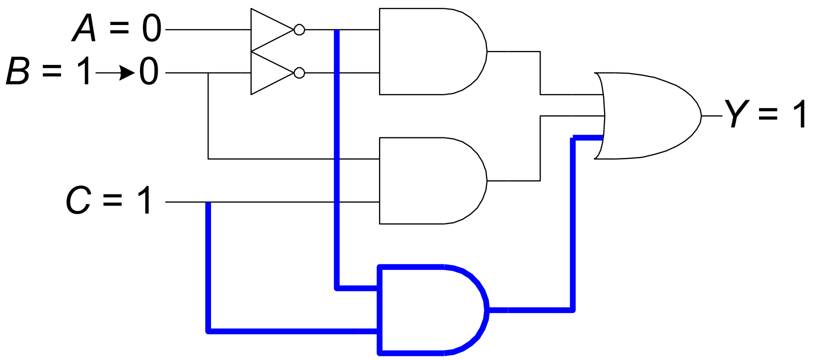
*For A=0 and C=1 --> Y* $\propto$ $\overline{B}$ *+ B*
*so if B changes it causes a glitch*

# Fixing the glitch



| C \ AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 |

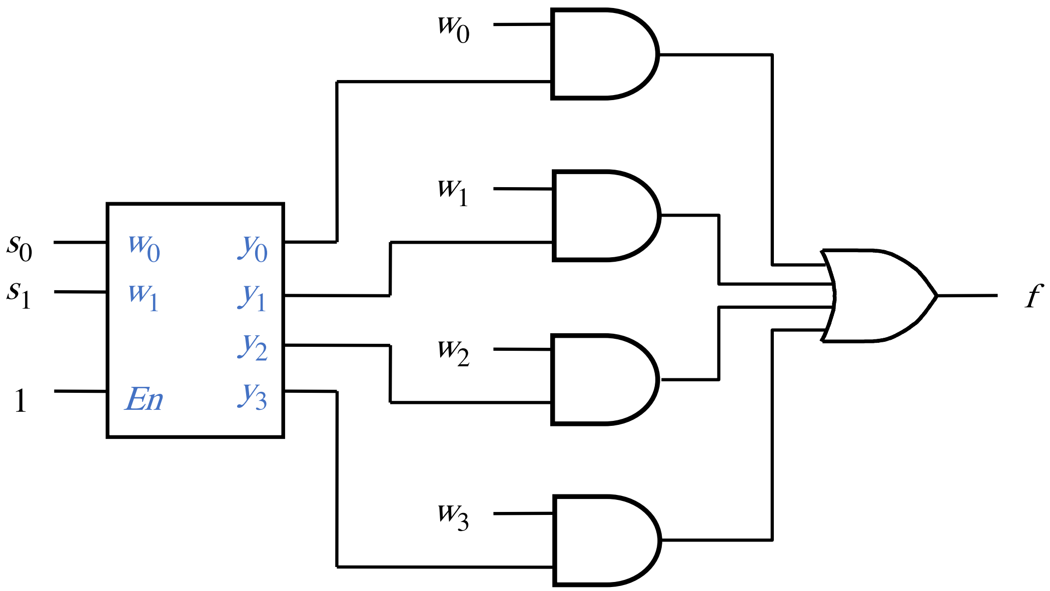
$Y = \bar{A}\bar{B} + BC + \bar{A}C$



# Next Time

- Basic sequential logic elements (latches and flip-flops)

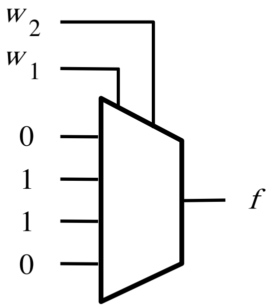
# Example



4-to-1 multiplexer built using a decoder.

# Examples of logic functions using muxes: 2-input XOR

| $w_1$ | $w_2$ | $f$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

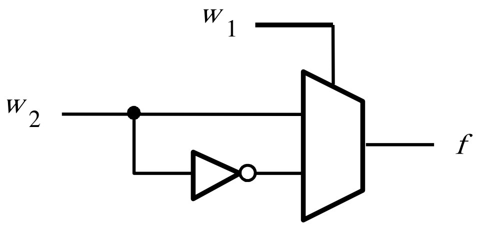


Implementation using a 4-to-1 multiplexer

| $w_1$ | $w_2$ | $f$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

| $w_1$ | $f$ |
|---|---|
| 0 | $w_2$ |
| 1 | $\overline{w}_2$ |

Modified truth table



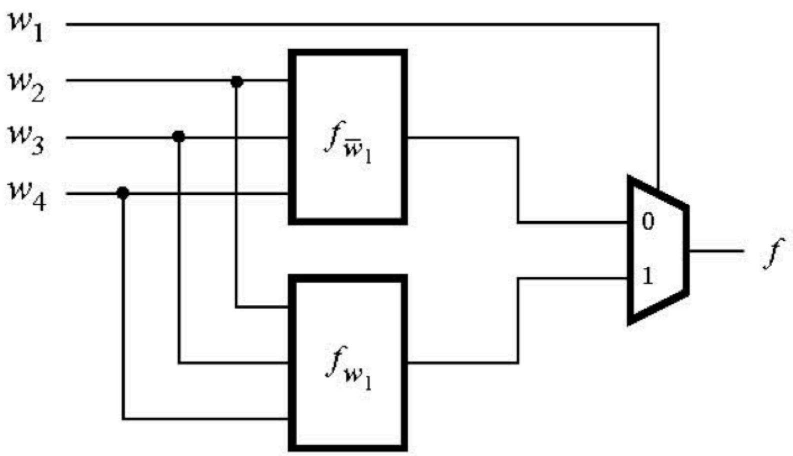
Implementation using a 2-to-1 multiplexer and a NOT

# Using muxes to build logic functions

- The key is to use Shannon’s expansion

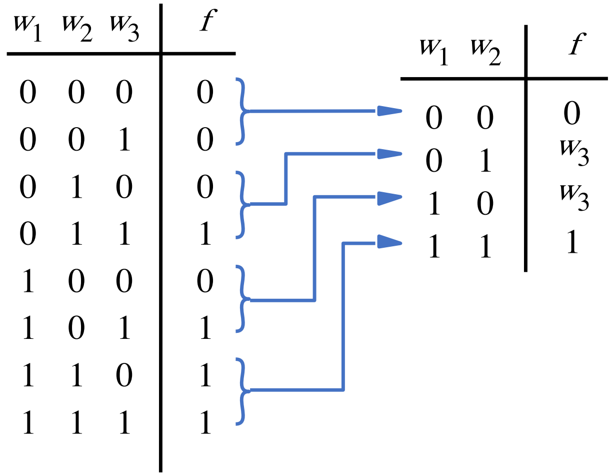
*Example:*

$$f(w1, w2, w3, w4) = \overline{w1} \cdot \underbrace{f(0, w2, w3, w4)}_{= f_{\overline{w1}}} + w1 \cdot \underbrace{f(1, w2, w3, w4)}_{= f_{w1}}$$



*This strategy is used extensively inside FPGAs*

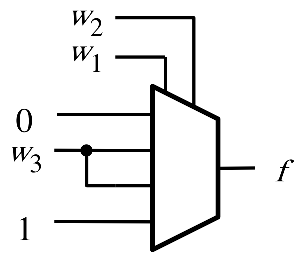
# Three input majority function using a 4:1 mux



| $w_1$ | $w_2$ | $w_3$ | $f$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

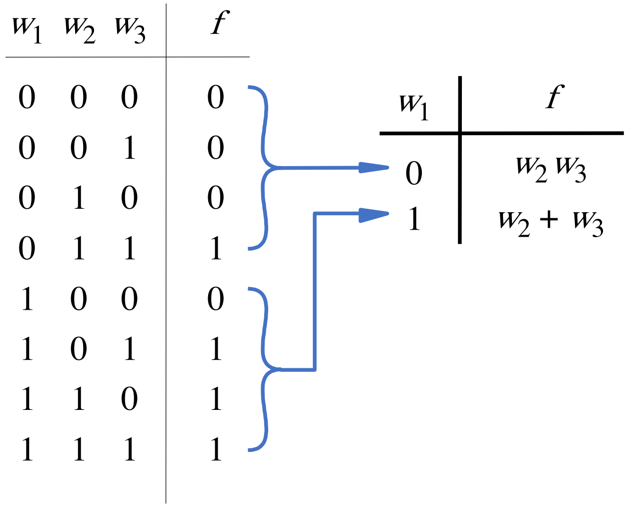
| $w_1$ | $w_2$ | $f$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | $w_3$ |
| 1 | 0 | $w_3$ |
| 1 | 1 | 1 |

(a) Modified truth table



(b) Circuit

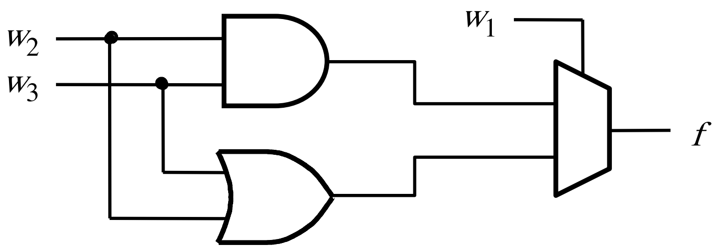
# Three-input majority function using a 2:1 mux



| $w_1$ | $w_2$ | $w_3$ | $f$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

| $w_1$ | $f$ |
|---|---|
| 0 | $w_2 w_3$ |
| 1 | $w_2 + w_3$ |

(b) Truth table



(b) Circuit

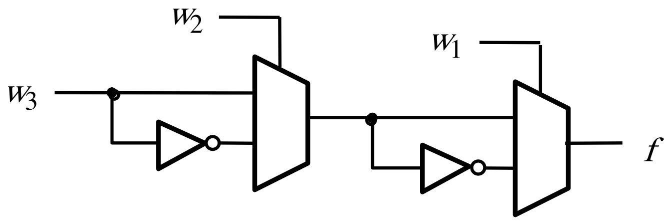
# Three-input XOR using 2:1 muxes

| $w_1$ | $w_2$ | $w_3$ | $f$ |
|---|---|---|---|
| 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 |

$w_2 \oplus w_3$

$\overline{w_2 \oplus w_3}$

(a) Truth table

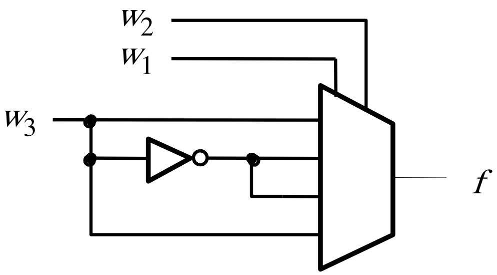


(b) Circuit

# Three-input XOR using a 4:1 mux

| $w_1$ | $w_2$ | $w_3$ | $f$ | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | $w_3$ |
| 0 | 0 | 1 | 1 | |
| 0 | 1 | 0 | 1 | $\overline{w}_3$ |
| 0 | 1 | 1 | 0 | |
| 1 | 0 | 0 | 1 | $\overline{w}_3$ |
| 1 | 0 | 1 | 0 | |
| 1 | 1 | 0 | 0 | $w_3$ |
| 1 | 1 | 1 | 1 | |

(a) Truth table



(b) Circuit

# S.L. Basic Elements (Latches & Flip-Flops)

CPEN 230 – Introduction to Digital Logic

# Review: CL Building Blocks and Delays

- Adders
- Multiplexers
- Decoders
- Demultiplexer
- Encoders
- Priority Encoder
- Comparator

- Propagation Delay
- Critical and Short Path
- Glitches

# One more C.L. design example:
# Binary-Coded-Decimal (BCD) Representation

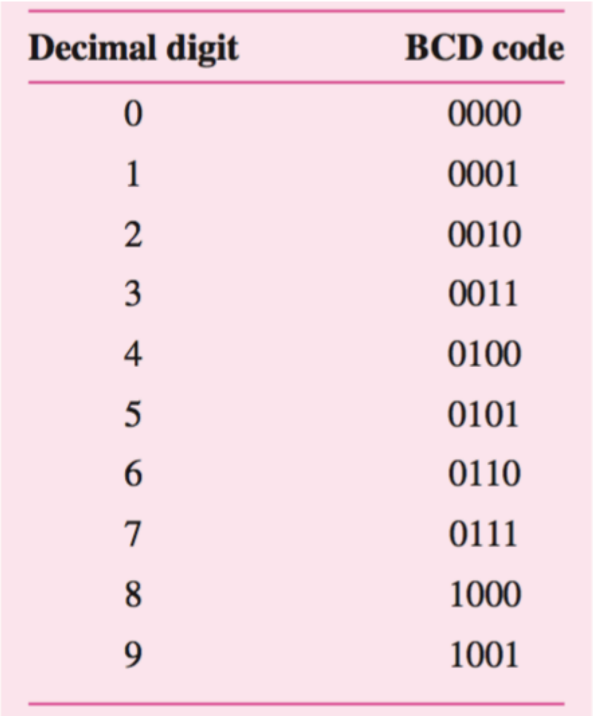
- BCD is an intermediate representation between Binary and Decimal
- It represents decimal numbers by encoding each decimal digit in binary form.
- Because there are 10 digits to represent we need four bits.

- Example: $58_{10} = 0101\ 1000_{BCD}$

# Binary-Coded-Decimal Representation

- BCD provides a format that is convenient when numerical information is to be displayed on a simple digit-oriented display

- BCD representation was used in some early computers and many handheld calculators



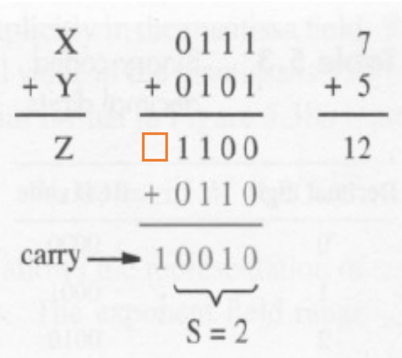
| Decimal digit | BCD code |
|---|---|
| 0 | 0000 |
| 1 | 0001 |
| 2 | 0010 |
| 3 | 0011 |
| 4 | 0100 |
| 5 | 0101 |
| 6 | 0110 |
| 7 | 0111 |
| 8 | 1000 |
| 9 | 1001 |

# BCD Addition

- The addition of two BCD digits is complicated by the fact that the sum may exceed 9, in which case a correction will have to be made
  - if $X + Y \leq 9$, the addition is the same as the addition of 2 four bit unsigned binary numbers
  - if $X + Y > 9$, then the result requires two BCD digits
- There are two cases where a correction has to be made
  1) The sum is greater than 9 but no carry-out is generated out of the four bits
  2) The sum is greater than 15 so a carry-out is generated out of the four bits

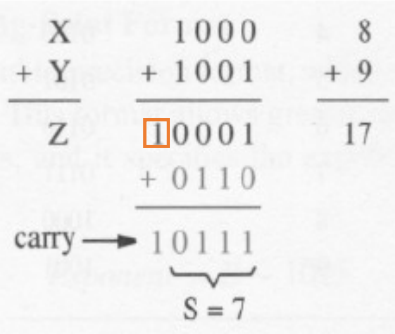
# BCD Addition – example

*9 < Z ≤ 15 → no carry*



```
   X       0111      7
 + Y     + 0101    + 5
 ----    ------    ---
   Z      □1100     12
         + 0110
         ------
carry →  10010
          S = 2
```

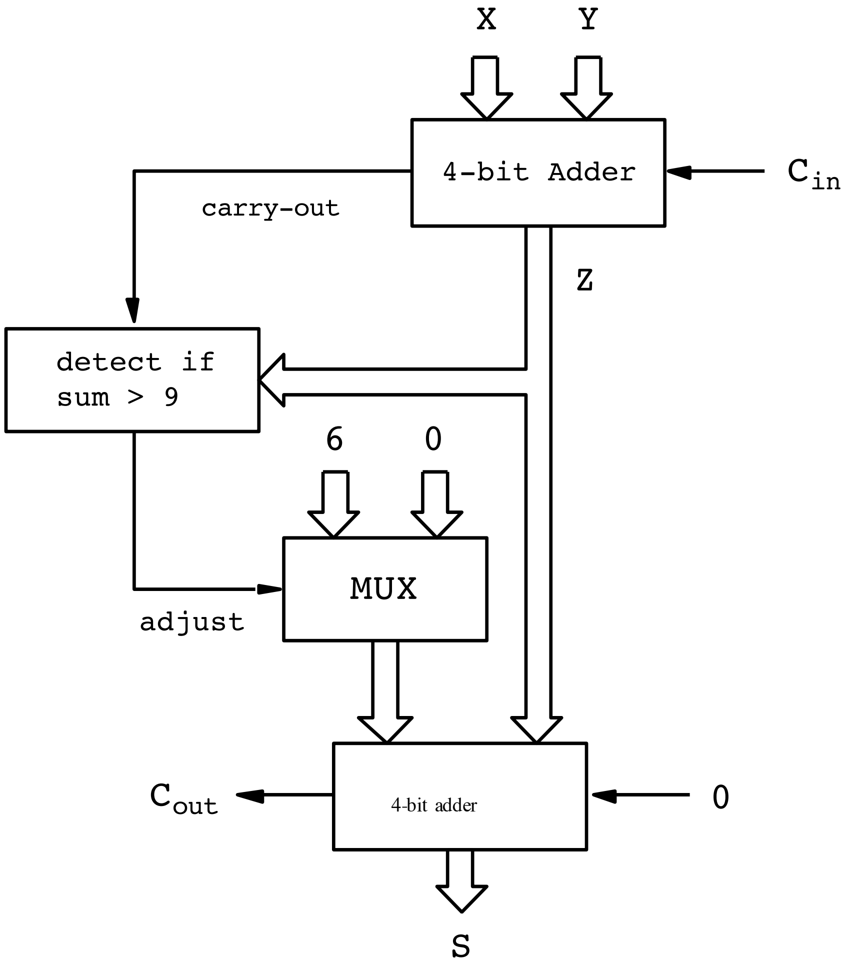
*Z > 15 → carry*



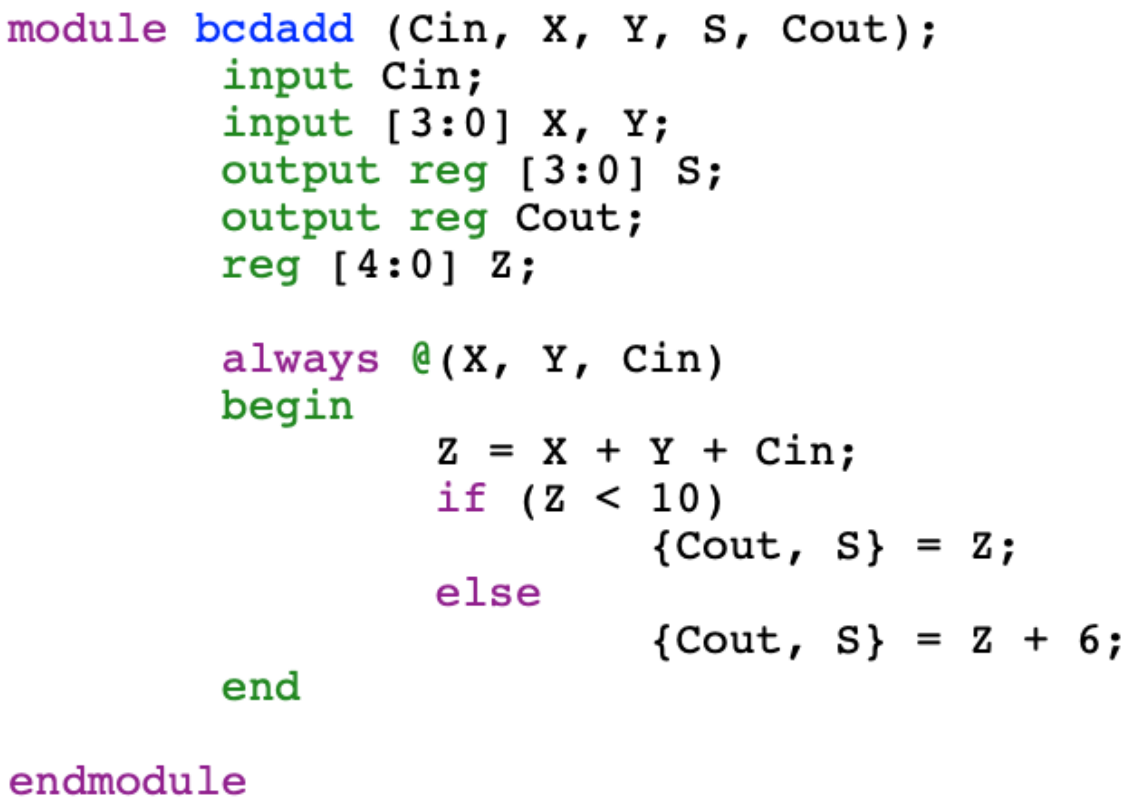
```
   X       1000      8
 + Y     + 1001    + 9
 ----    ------    ---
   Z      10001     17
         + 0110
         ------
carry →  10111
          S = 7
```

Whenever the result of the 4-bit addition (Z) exceeds 9, the correct decimal digit can be generated by adding 6 to the result.

# One-Digit BCD Adder

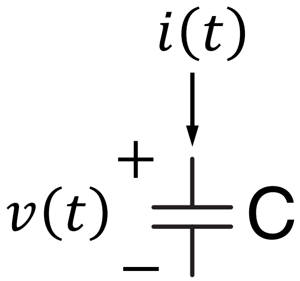


# Verilog Code for a One-Digit BCD Adder



```
module bcdadd (Cin, X, Y, S, Cout);
        input Cin;
        input [3:0] X, Y;
        output reg [3:0] S;
        output reg Cout;
        reg [4:0] Z;

        always @(X, Y, Cin)
        begin
                Z = X + Y + Cin;
                if (Z < 10)
                        {Cout, S} = Z;
                else
                        {Cout, S} = Z + 6;
        end

endmodule
```

# Present and Past



$$i(t) = C\frac{dv(t)}{dt} \overset{\substack{\lim\limits_{\Delta t\to 0}\\ \downarrow}}{\cong} C\frac{\Delta v}{\Delta t} \cong C\frac{\overset{\substack{\textit{present}\\ \downarrow}}{v(t)} - \overset{\substack{\textit{past}\\ \downarrow}}{v(t_0)}}{t - t_0}$$

$$q(t) = C \cdot v(t)$$

$$dq(t) = C \cdot dv(t)$$

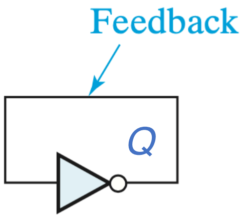
Capacitors stores charge (i.e. voltage level *v=q/C*).
Ideally the charge should be stored forever,
but, … unfortunately in practical capacitors
the change is not hold forever, eventually it leaks.

# Sequential Circuits

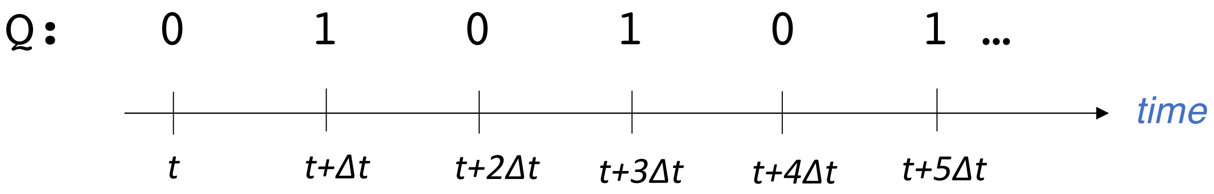
- Recall that a combinational circuit produces an output that depends only on the current state of its input --> combinational circuits must be acyclic (no loops).
- If we add a feedback to a combinational circuit, the circuit becomes sequential.
- The output of a sequential circuit depends not only on its current input, but also on the history of its previous inputs. The cycle created by the feedback allows the circuit to store information (i.e. remember) about its previous input.
- We call the information “stored” on the feedback signal as the state of the circuit.

# Adding Feedback

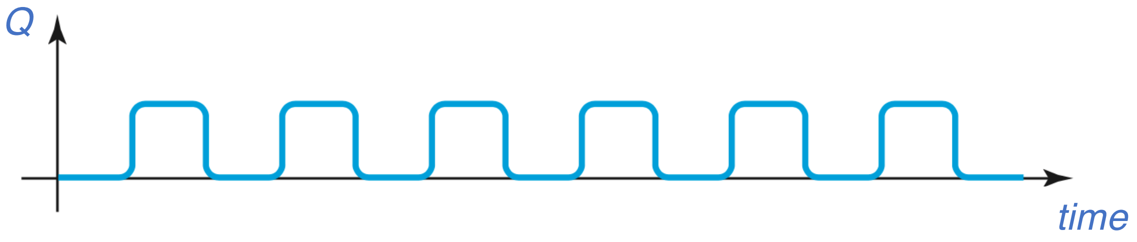
- Easiest option we can think of is to wrap feedback around an inverter



(a) Inverter with feedback



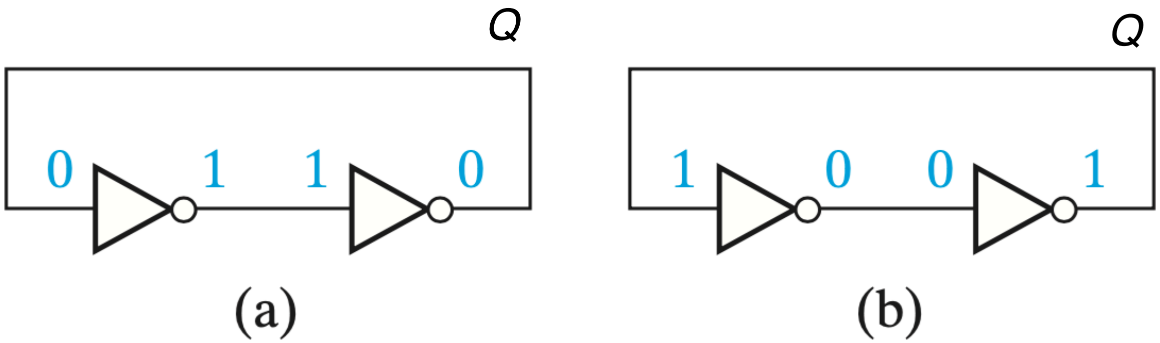
- The inverter output will continue to oscillate back and forth between 1 and 0, and it will never  reach a stable condition. The rate at which the circuit oscillates is determined by the propagation delay Δt of the inverter.



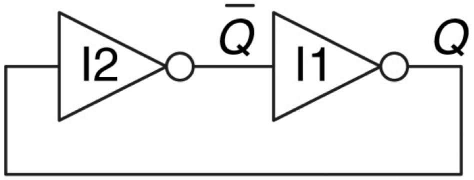
(b) Oscillation at inverter output

## Adding Feedback

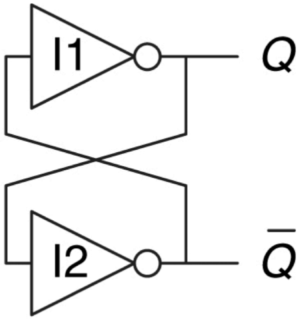
- Wrapping feedback around an inverter causes the state to keep flipping. Not quite what we are looking for. What we are looking for is holding (remembering) the state in a stable manner.
- … adding one more inverter does the trick



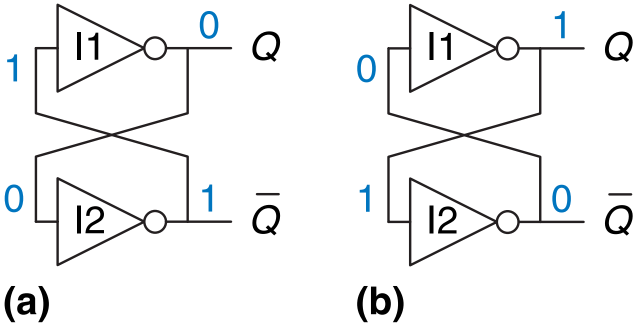
# Feedback is Remembering



*Analyzing this circuit is different from analyzing a combinational circuit, because it is cyclic: Q depends on $\overline{Q}$ and $\overline{Q}$ depends on Q*



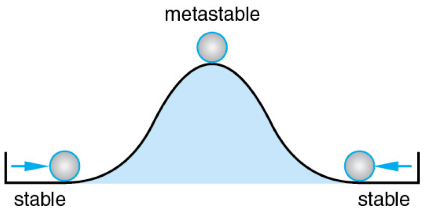
Just as *Y* is commonly used for the output of combinational logic, *Q* is commonly used for the output of sequential logic.



*To emphasize that the two inverters are cross-coupled the circuits is commonly redrawn as shown above.*

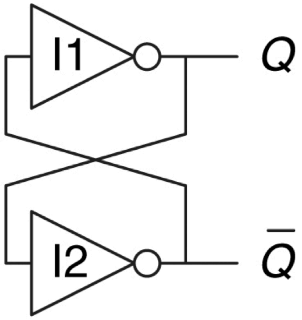
*case I. Q = 0 ⟼ the cell holds the value “forever” (its “state” is stable)*

*case II. Q = 1 ⟼ the cell holds the value “forever” (its “state” is stable)*



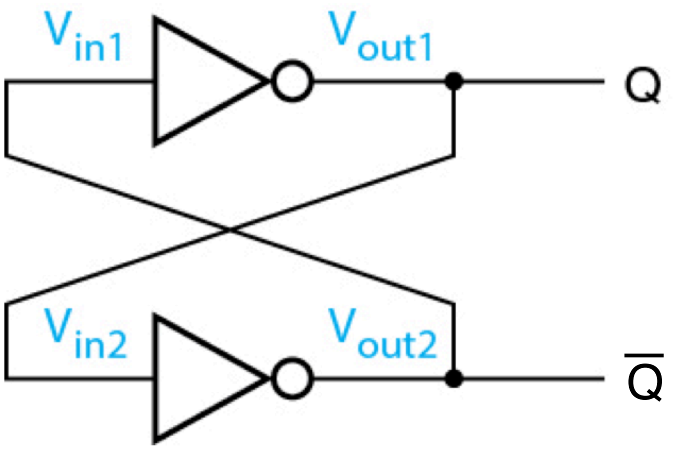
*Because the circuit has two stable states Q=0 and Q=1, the circuit is said to be* ***bistable****.*

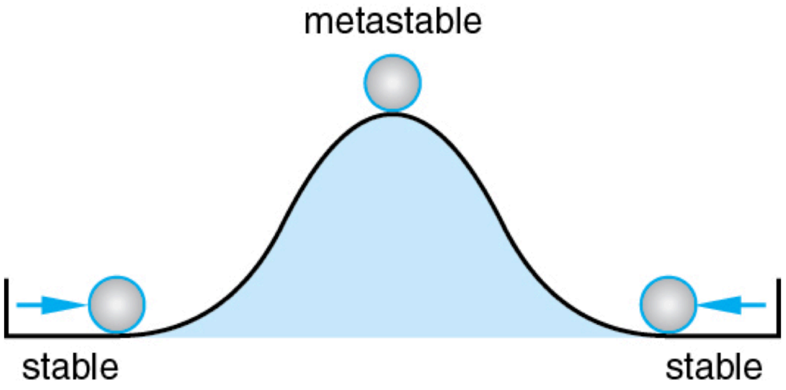
# Cross-coupled inverters

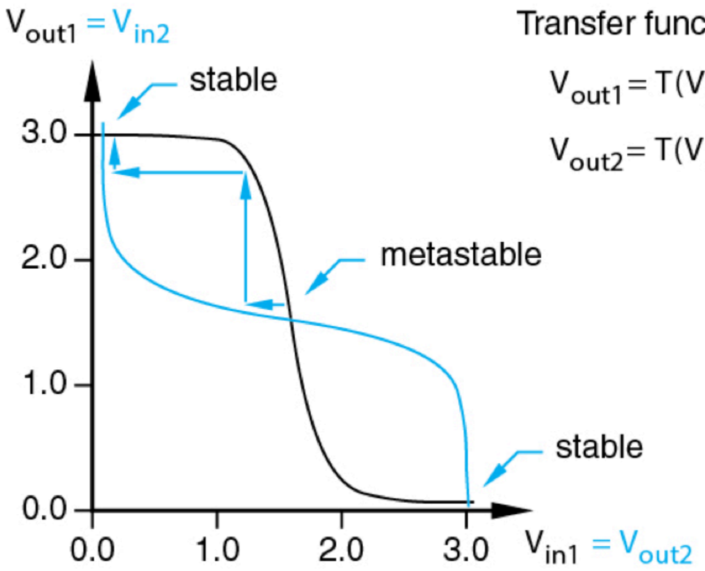


- Although the cross-coupled inverters can store a bit of information they are not practical because the user has no inputs to control the state.

# Cross-coupled inverters behavior





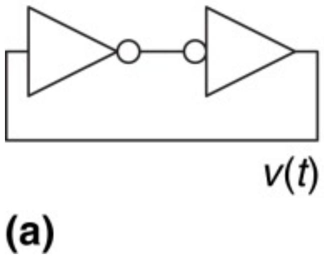


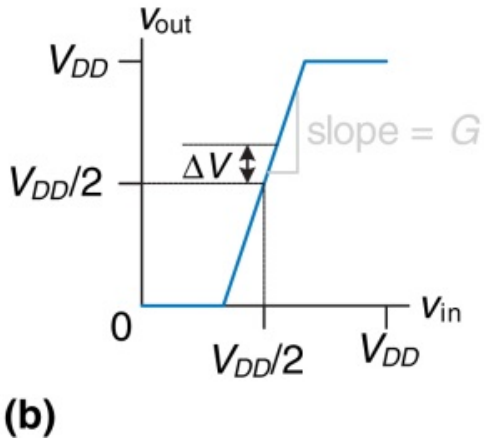
Transfer functions:

$V_{out1} = T(V_{in1})$

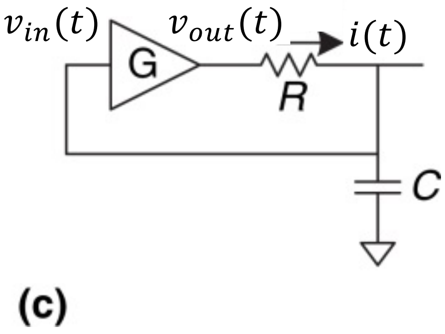
$V_{out2} = T(V_{in2})$

# Circuit model of cross-coupled bistable.





There is delay as the signal propagates through the loop



$$i(t) = \frac{v_{out}(t) - v_{in}(t)}{R}$$

$$v_{out}(t) = G \cdot v_{in}(t)$$

$$i(t) = C\frac{dv_{in}(t)}{dt}$$

$$v_{out}(0) = \Delta V$$

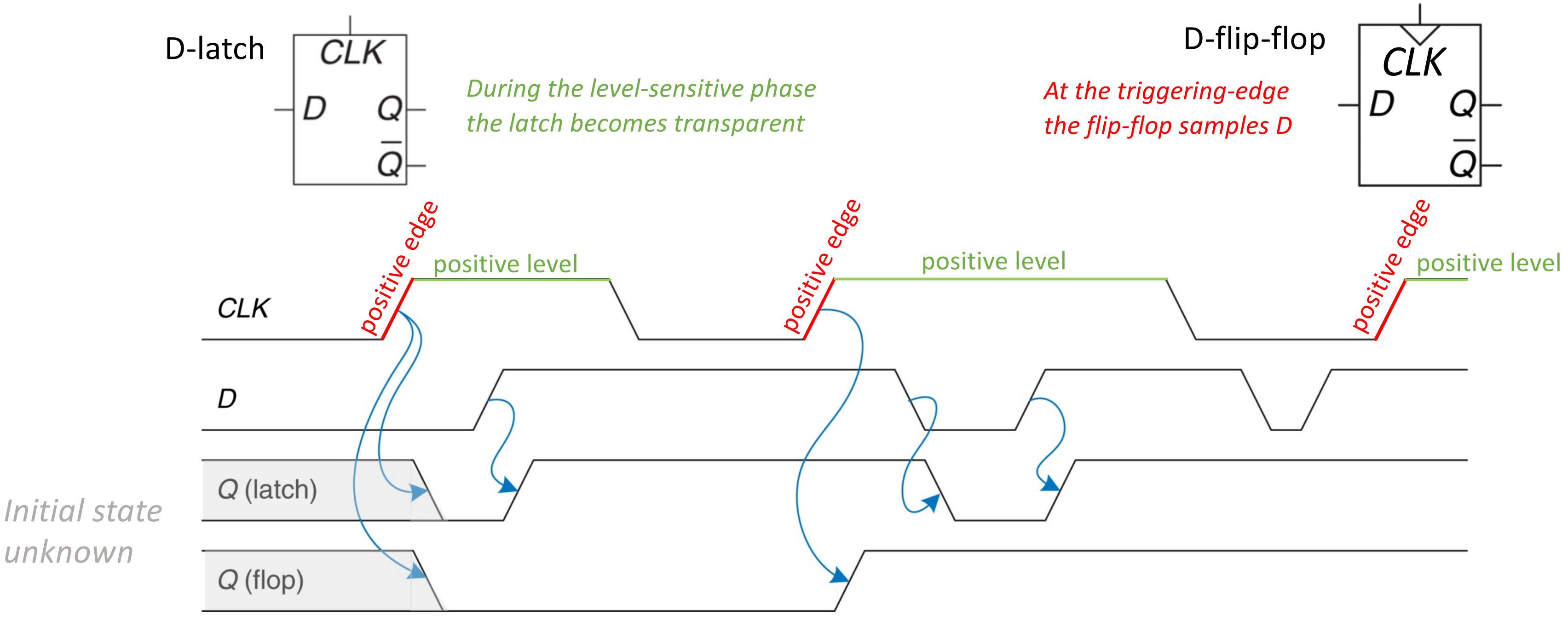
$$\Longrightarrow \quad \frac{G-1}{RC}dt = \frac{dv_{out}}{v_{out}} \Leftrightarrow v_{out}(t) = \Delta V \cdot exp\left[\frac{(G-1)t}{RC}\right]$$

# Sequential circuits

- When power is first applied to a sequential circuit, the initial state is unknown and usually unpredictable. It may differs each time the circuit is turned on.

# Latches vs. Flip-Flops

- There are two types of storage elements: latches and flip-flops
  - A latch is a 1-bit level-sensitive storage element
  - A flip-flop is a 1-bit edge-triggered storage element



*During the level-sensitive phase the latch becomes transparent*

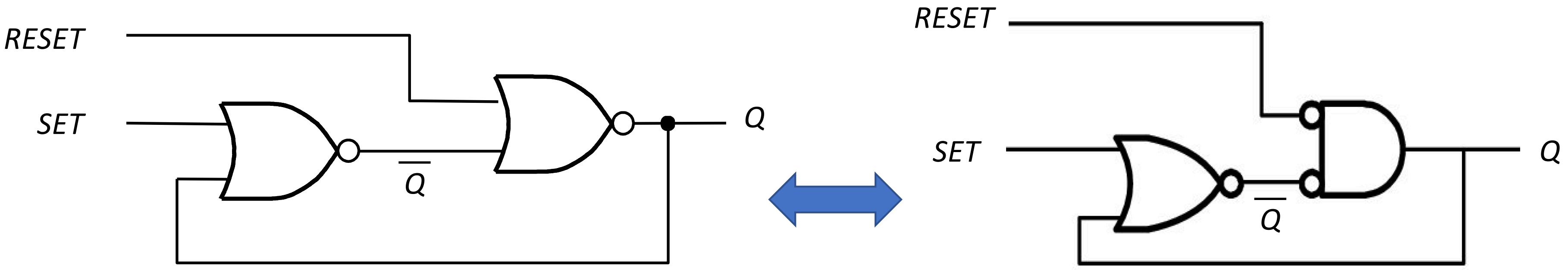
*At the triggering-edge the flip-flop samples D*

*Initial state unknown*

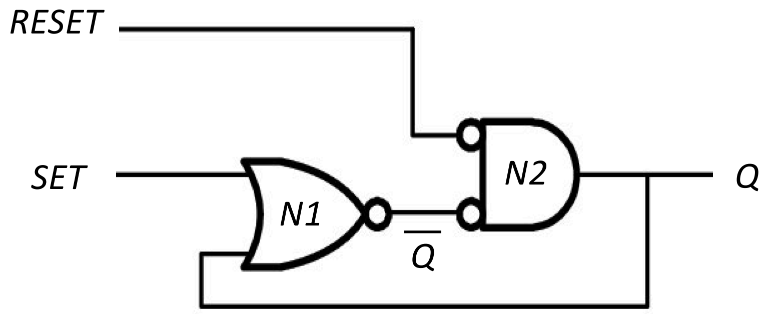
# What’s inside a D-latch ? What about a D flip-flop?

- Typical “introduction to Digital Logic’s textbook” approach.
  - Let’s wrap feedback around our favorite logic gate (e.g. NOR) and find a way to provide the capability of controlling the value (1/0) we want to store.

*If you like the NAND better apply DeMorgan*



# S-R latch



*case I.* ***R = 1 and S = 0***

since R=1 --> N2 gives Q=0 reset the output

since S=0 and Q=0 --> N1 gives Q'=1

*case II.* ***R = 0 and S = 1***

since S=1 --> N1 gives Q'=0

since Q'=0 --> N2 gives Q=1 set the output

*case III.* ***R = 1 and S = 1***

since S=1 --> N1 gives Q'=0

since R=1 --> N2 gives Q=0

invalid Q ≠ Q'

*case IV.* ***R = 0 and S = 0***

since S=O --> we cannot tell what N1 gives unless we know Q

since R=0 --> we cannot tell what N2 gives unless we know Q'

*case IVa.* ***R = 0 and S = 0 and Q=0***

since S=O and Q=0 --> N1 gives Q'=1

since R=0 (and Q'=1) --> N2 gives Q=0

*case IVb.* ***R = 0 and S = 0 and Q=1***

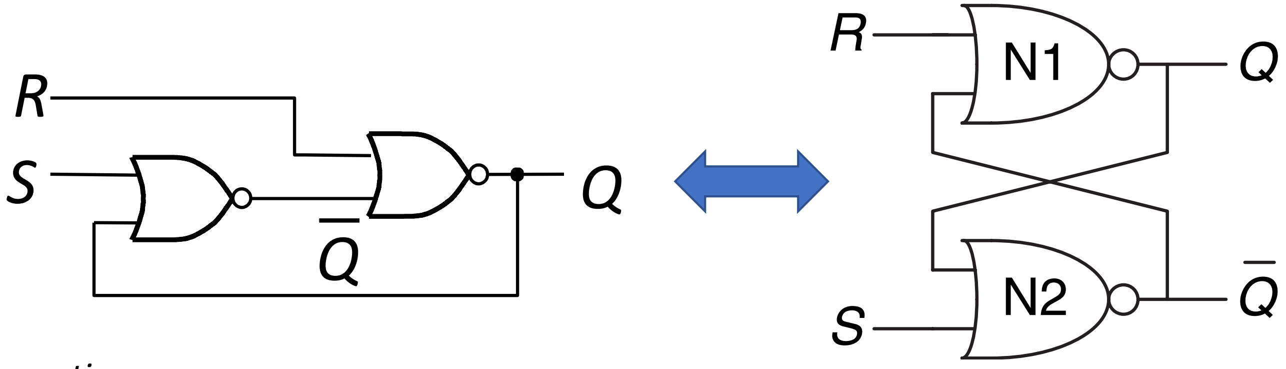
since S=O and Q=1 --> N1 gives Q'=0

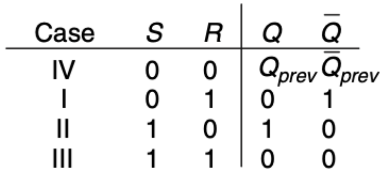
since R=0 (and Q'=0) --> N2 gives Q=1

$Q = Q_{prev}$

Memory !

# Summary of SR latch results





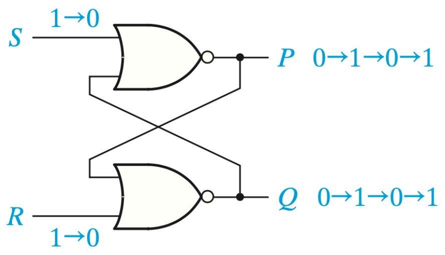
| Case | $S$ | $R$ | $Q$ | $\overline{Q}$ |
|---|---|---|---|---|
| IV | 0 | 0 | $Q_{prev}$ | $\overline{Q}_{prev}$ |
| I | 0 | 1 | 0 | 1 |
| II | 1 | 0 | 1 | 0 |
| III | 1 | 1 | 0 | 0 |

**Figure 3.5** **SR latch truth table**

*Observation*:

in case III (R=S=1) the node Q' is equal to 0 just like Q (in all other cases the node Q' was the complement of Q). This issue could be swept under the rag. The output of N1 doesn't have to be necessarily the complement of Q, we can call the node P and do not use it as an output of the cell (so nobody has to know). However there is a worst issue that cannot be ignored, and doesn't not become evident unless we analyze the circuit dynamically. Suppose after R=S=1 the inputs both goes simultaneously to R=S=0 and assume the delays of the two gates are identical: the circuit start to oscillate.

**FIGURE**
Improper S-R Latch
Operation



# SR latch operation

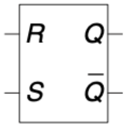
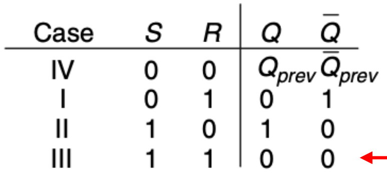


**Figure 3.6** **SR latch symbol**



| Case | $S$ | $R$ | $Q$ | $\overline{Q}$ |
|---|---|---|---|---|
| IV | 0 | 0 | $Q_{prev}$ | $\overline{Q}_{prev}$ |
| I | 0 | 1 | 0 | 1 |
| II | 1 | 0 | 1 | 0 |
| III | 1 | 1 | 0 | 0 |

*Invalid operation we must avoid case III*

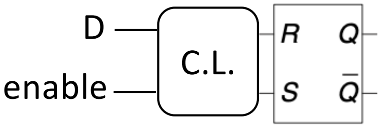
**Figure 3.5** **SR latch truth table**

Sometime people prefer to call the truth tables for S.L. with the name characteristic tables (to emphasize they are not talking about C.L.)

# The Big Picture:

At the end of the day what we want is either set the state to a new value (either 0 or 1) or hold the existing state (and this can all be achieved through cases I, II, and IV).

```
if enable == 1  Q <= D ;  // specify the value of the new state
else Q <= Qprev;          // hold the new state to its old value
```

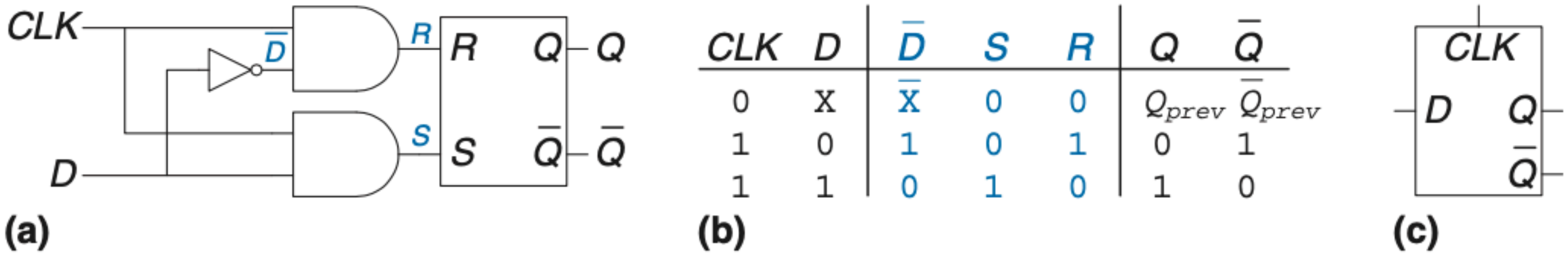


**Toward the D-latch**

| enable D | S R |
|---|---|
| 0 0 | 0 0 |
| 0 1 | 0 0 |
| 1 0 | 0 1 |
| 1 1 | 1 0 |

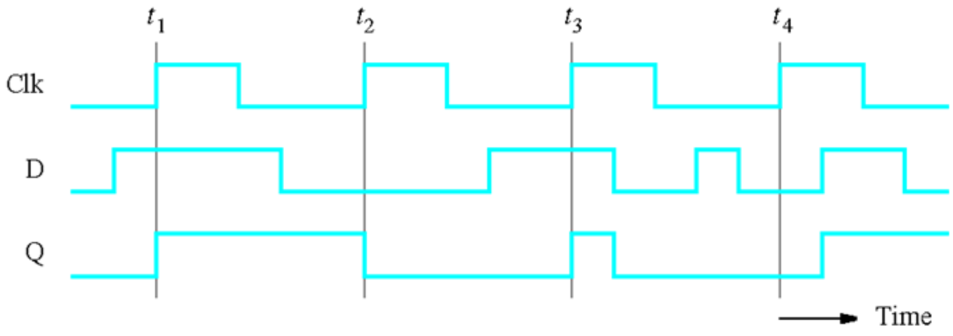
$$S = enable \cdot D$$
$$R = enable \cdot \overline{D}$$

# D latch



| CLK | D | $\overline{D}$ | S | R | Q | $\overline{Q}$ |
|---|---|---|---|---|---|---|
| 0 | X | $\overline{X}$ | 0 | 0 | $Q_{prev}$ | $\overline{Q}_{prev}$ |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 |

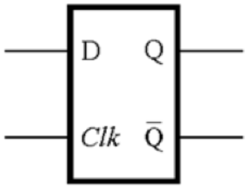
**Figure 3.7** **D latch: (a) schematic, (b) truth table, (c) symbol**



(d) Timing diagram

# Common conventions to write the characteristic tables

*Example: D-latch*



| CLK | D | Q |
|---|---|---|
| 0 | 0 | $Q_{prev}$ |
| 0 | 1 | $Q_{prev}$ |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| CLK | D | $Q_{new}$ |
|---|---|---|
| 0 | 0 | $Q_{old}$ |
| 0 | 1 | $Q_{old}$ |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| CLK | D | Q(t+1) |
|---|---|---|
| 0 | 0 | Q(t) |
| 0 | 1 | Q(t) |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| CLK | D | $Q_{n+1}$ |
|---|---|---|
| 0 | 0 | $Q_n$ |
| 0 | 1 | $Q_n$ |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

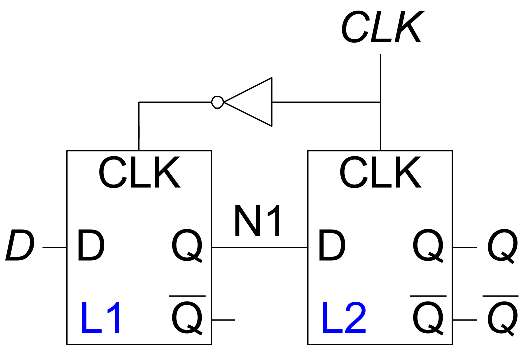
| CLK | D | $Q_{next}$ |
|---|---|---|
| 0 | 0 | Q |
| 0 | 1 | Q |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| CLK | D | $Q^+$ |
|---|---|---|
| 0 | 0 | Q |
| 0 | 1 | Q |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| CLK | D | Q | $Q_{next}$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

# D flip-flop internal circuit

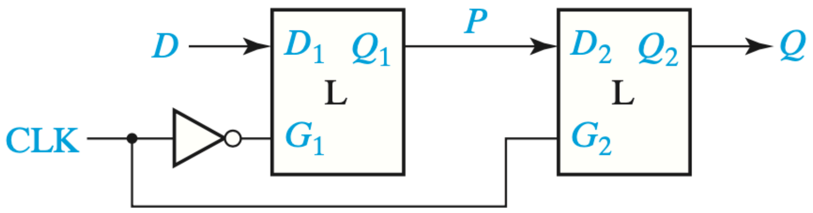
- A D flip-flop can be built from 2 back to back D latches controlled by complementary clocks. The first latch L1 is called the master. The second latch L2 is called the slave.



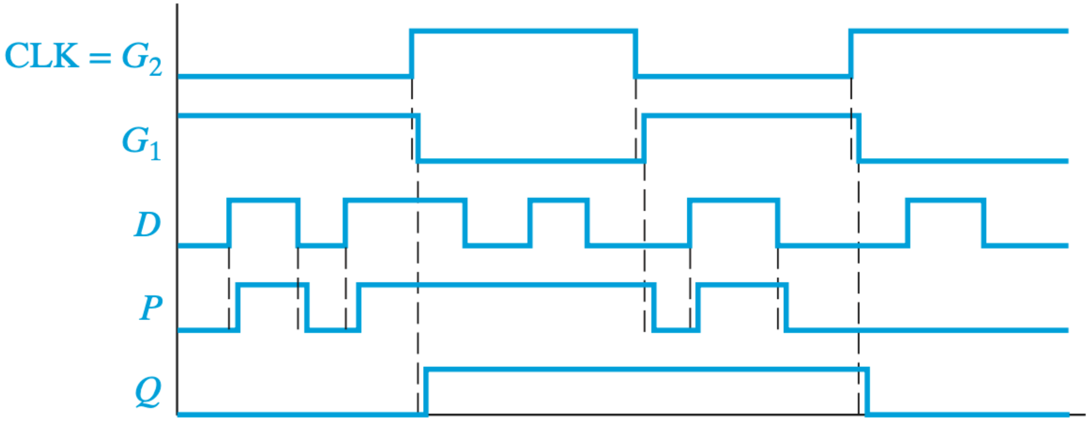
- When ***CLK* = 0**
  - L1 is transparent
  - L2 is opaque
  - *D* passes through to N1
- When ***CLK* = 1**
  - L2 is transparent
  - L1 is opaque
  - N1 passes through to *Q*
- Thus, on the edge of the clock (when ***CLK* rises from 0 to 1**)
  - *D* passes through to *Q*

# D Flip-Flop internal circuit

D Flip-Flop (Rising-Edge Trigger)

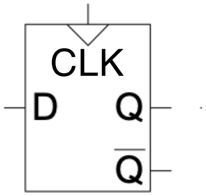


(a) Construction from two gated D latches



(b) Timing analysis

# D flip-flop



*DFF Symbol*

- Inputs: *CLK*, *D*
- Function
  - Samples *D* on rising edge of *CLK*
    - When *CLK* rises from 0 to 1, *D* passes through to *Q*
    - Otherwise, *Q* holds its previous value
  - *Q* changes only on rising edge of *CLK*
- Called *edge-triggered* *(i.e. activated on the clock edge)*

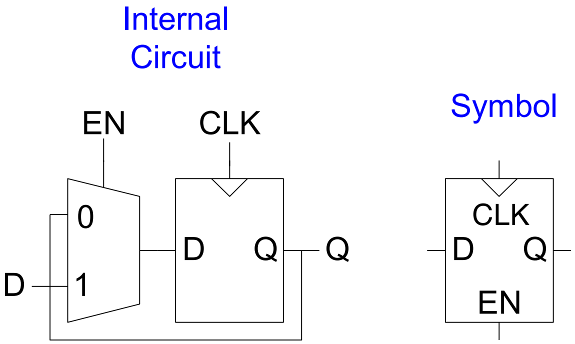
| *CLK* | *D* | *Q* | $Q^+$ |
|---|---|---|---|
| ↑ | 0 | 0 | 0 |
| ↑ | 0 | 1 | 0 |
| ↑ | 1 | 0 | 1 |
| ↑ | 1 | 1 | 1 |

(c) Truth table

$$\boxed{Q^+ = D}$$

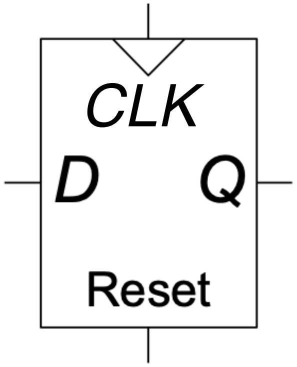
# Enabled Flip-Flops

- Inputs: *CLK*, *D*, *EN*
  - The enable input (*EN*) controls when new data (*D*) is stored
- Function
  - *EN* = 1: *D* passes through to *Q* on the clock edge
  - *EN* = 0: the flip-flop retains its previous state



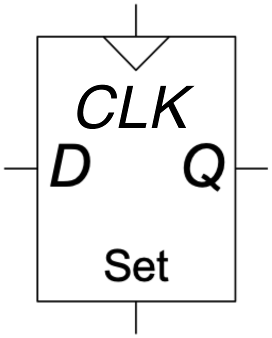
# Resettable Flip-Flops

- Inputs: *CLK*, *D*, *Reset*
- Function:
  - *Reset* = 1: *Q* is forced to 0
  - *Reset* = 0: flip-flop behaves as ordinary D flip-flop



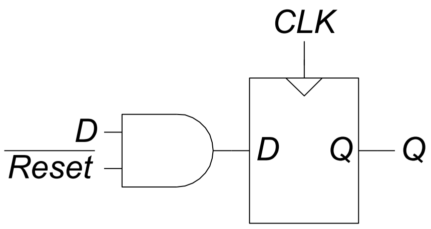
# Settable Flip-Flops

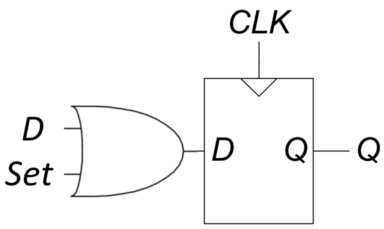
- Inputs: *CLK*, *D*, *Set*
- Function:
  - *Set* = 1: *Q* is forced to 1
  - *Set* = 0: flip-flop behaves as ordinary D flip-flop



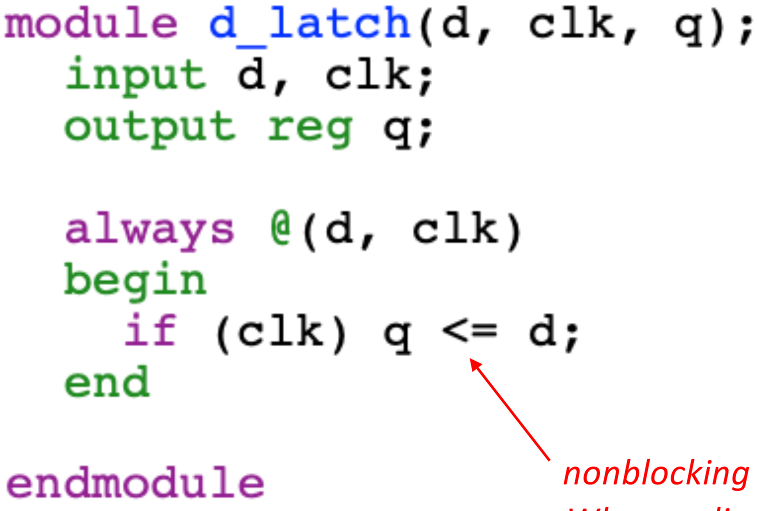
# Resettable/Settable Flip-Flops

- Two types:
  - Synchronous: resets/sets at the clock edge only
  - Asynchronous: resets/sets immediately when *Reset/Set* = 1
- Asynchronously resettable/settable flip-flop requires changing the internal circuitry of the flip-flop
- Synchronously resettable/settable flip-flop doesn’t.

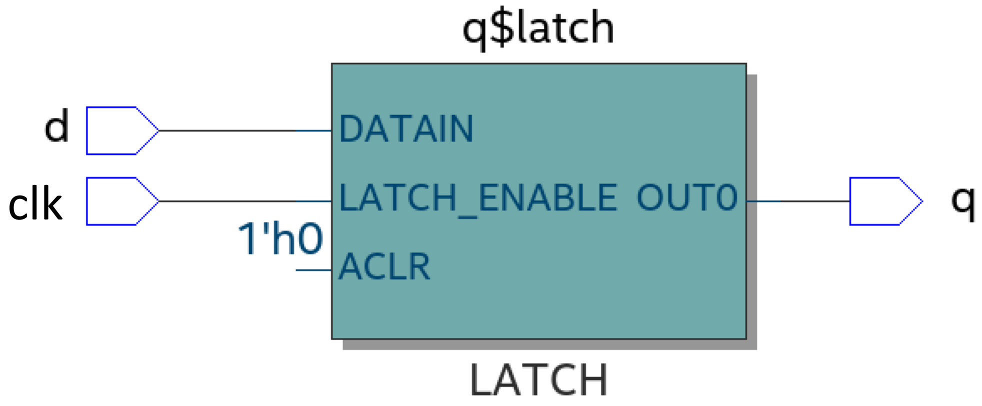




# D latch



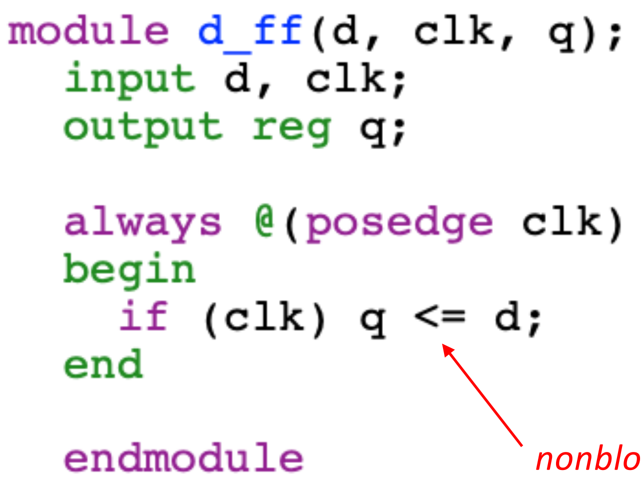
```
module d_latch(d, clk, q);
  input d, clk;
  output reg q;

  always @(d, clk)
  begin
    if (clk) q <= d;
  end

endmodule
```



*nonblocking assignment*

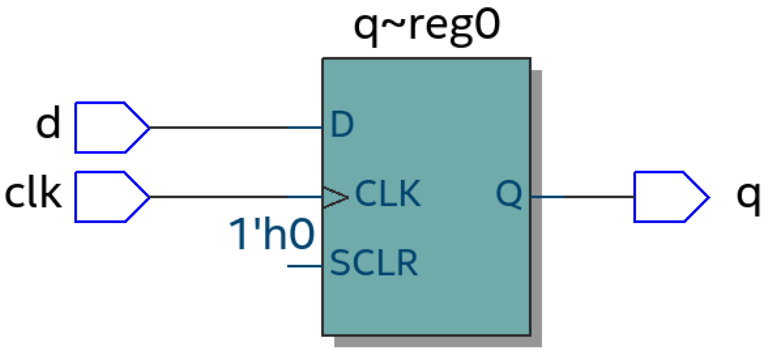
*When coding S.L. use only nonblocking assignment*

# D flip-flop



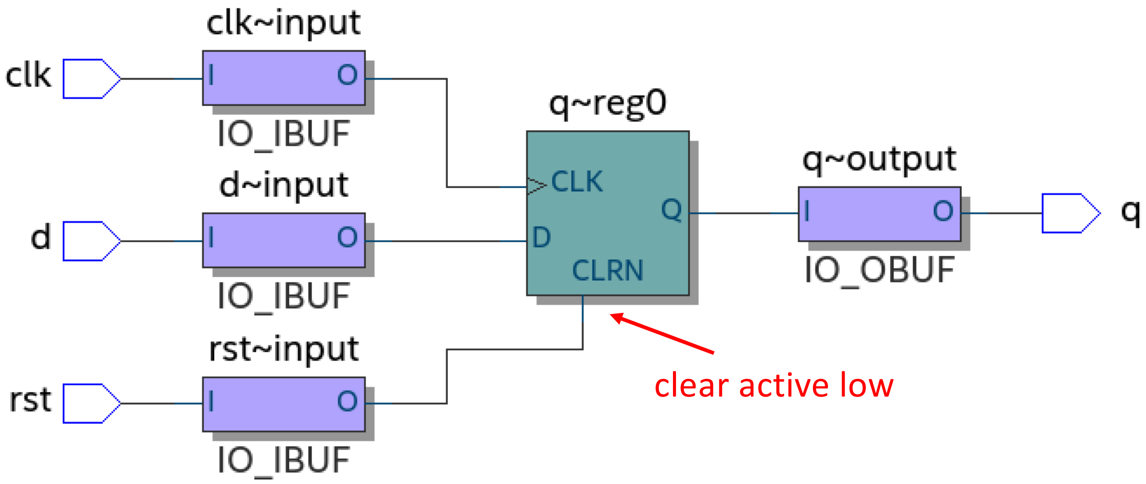
```
module d_ff(d, clk, q);
  input d, clk;
  output reg q;

  always @(posedge clk)
  begin
    if (clk) q <= d;
  end

  endmodule
```

*nonblocking assignment*



# D flip-flop with asynchronous reset (active low)

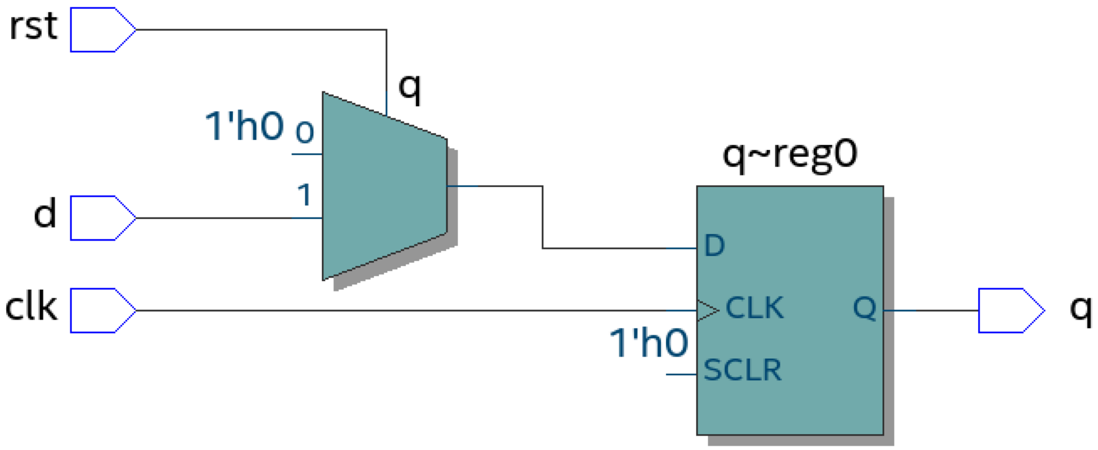
```
module dff_asy (d, rst, clk, q);
  input d, rst, clk;
  output reg q;

  always @(negedge rst, posedge clk)
  begin
    if (~rst)
      q <= 0;
    else
      q <= d;
  end

endmodule
```



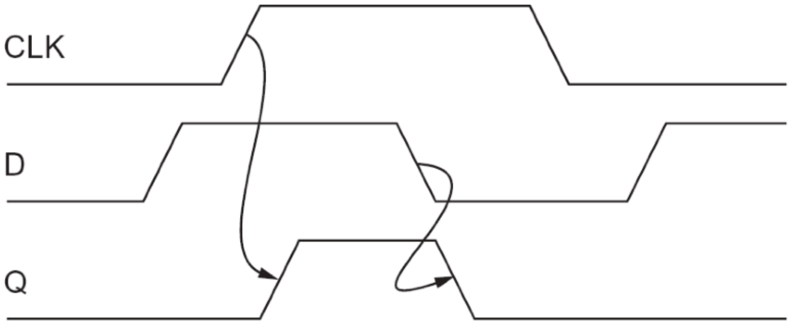
clear active low

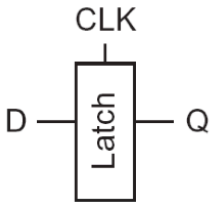
# D flip-flop with synchronous reset (active low)

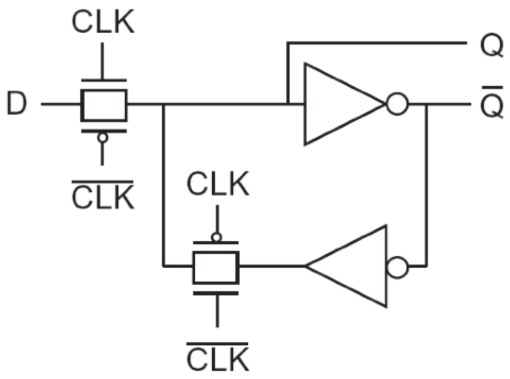
```
module dff_syn (d, rst, clk, q);
  input d, rst, clk;
  output reg q;

  always @(posedge clk)
    if (~rst) q <= 0;
    else q <= d;

endmodule
```



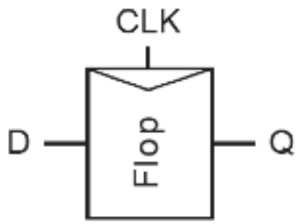
# Transistor level D-latch

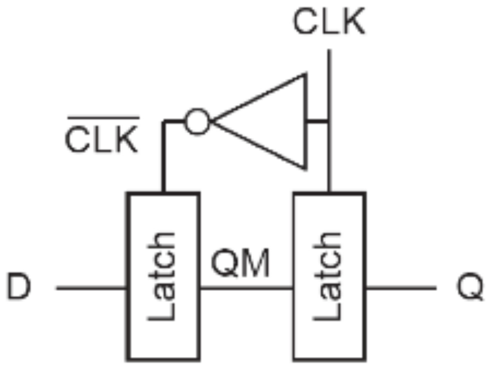


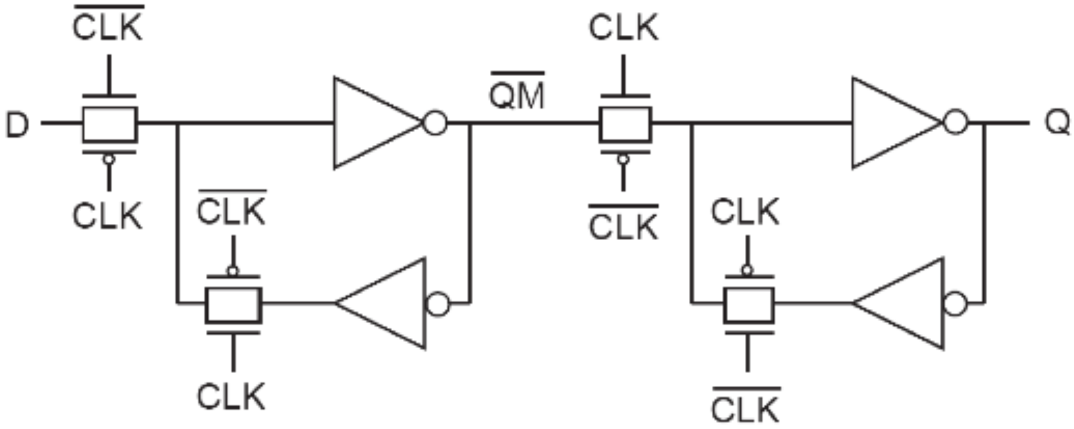


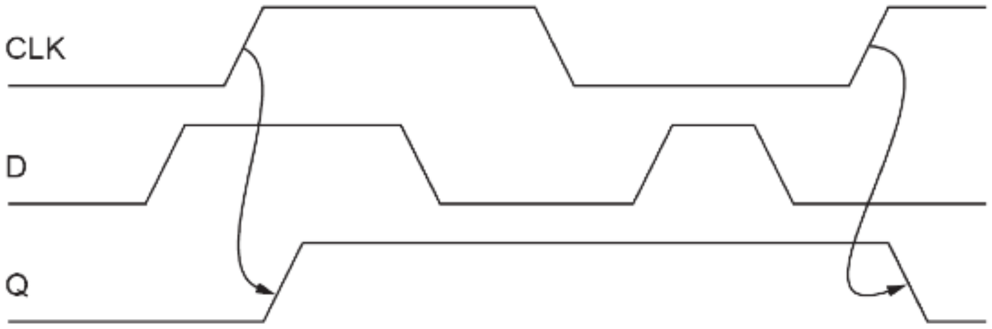


# Transistor level D flip-flop









# Next Time

- Sequential logic building blocks
    - JK Flip-Flop and T Flip-Flop
    - Registers
    - Shift Registers
    - Counters

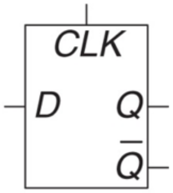
# S.L. Building Blocks

CPEN 230 – Introduction to Digital Logic

# Review: Latches and Flip-Flops

D-latch

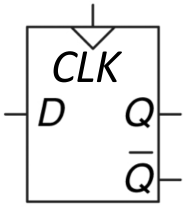
| CLK | D | Qn+1 |
|---|---|---|
| 0 | 0 | $Q_n$ |
| 0 | 1 | $Q_n$ |
| 1 | 0 | 0 |
| 1 | 1 | 1 |



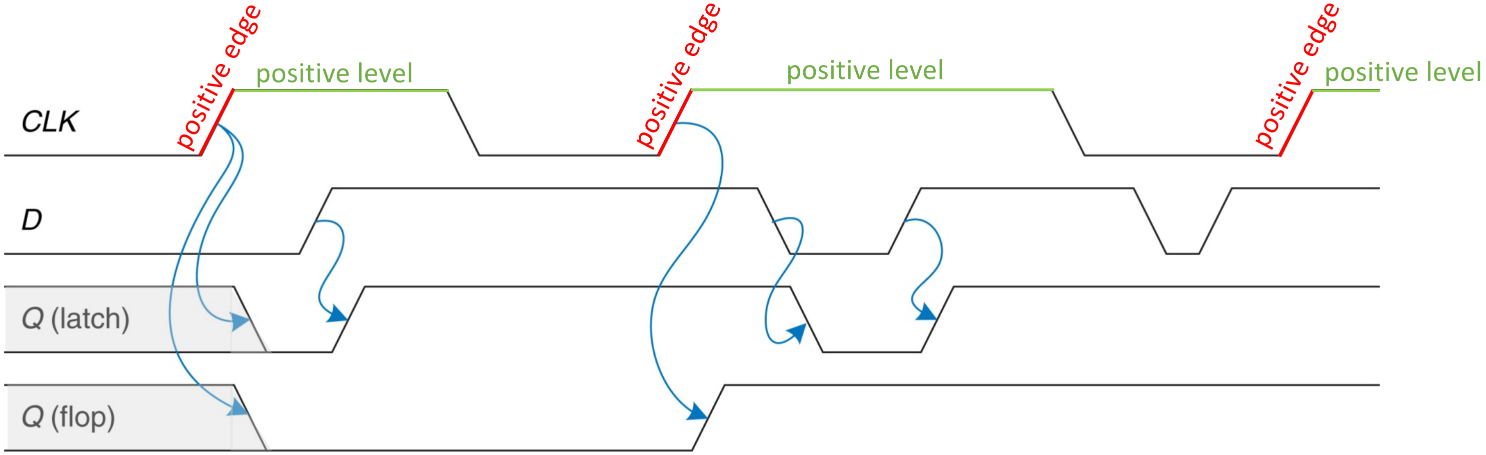
*During the level-sensitive phase the latch becomes transparent*

D-flip-flop

*At the triggering-edge the flip-flop samples D*



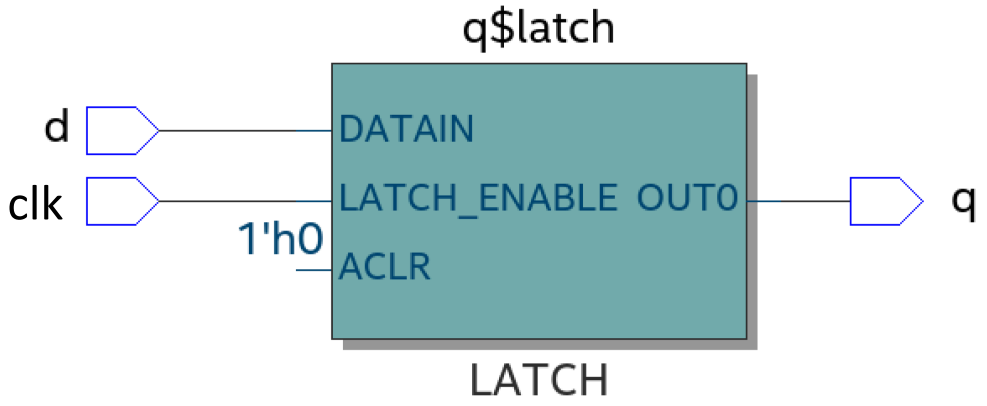
| CLK | D | Qn+1 |
|---|---|---|
| ↑ | 0 | 0 |
| ↑ | 1 | 1 |



*Initial state unknown*

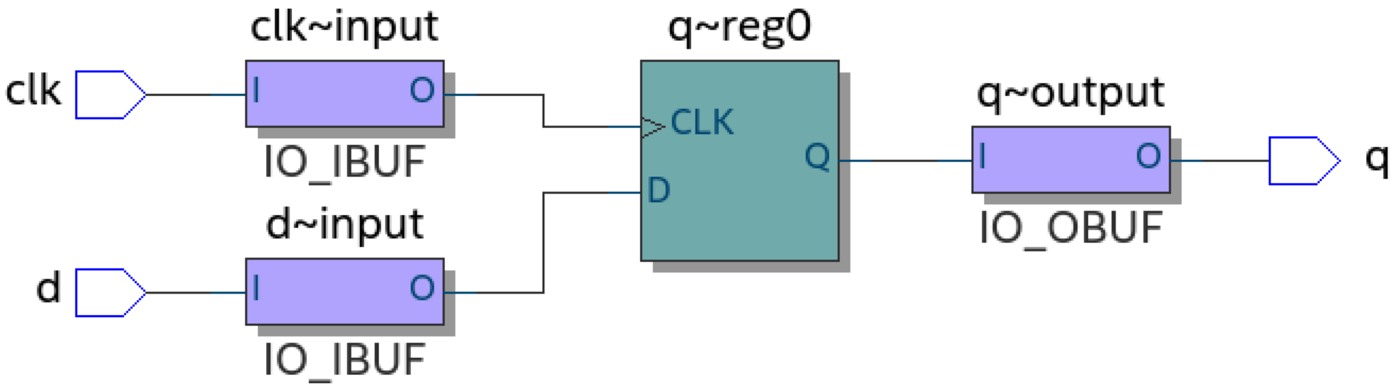
# Review: D latch in Verilog

```
module d_latch(d, clk, q);
  input d, clk;
  output reg q;

  always @(d, clk)
  begin
    if (clk) q <= d;
  end

endmodule
```



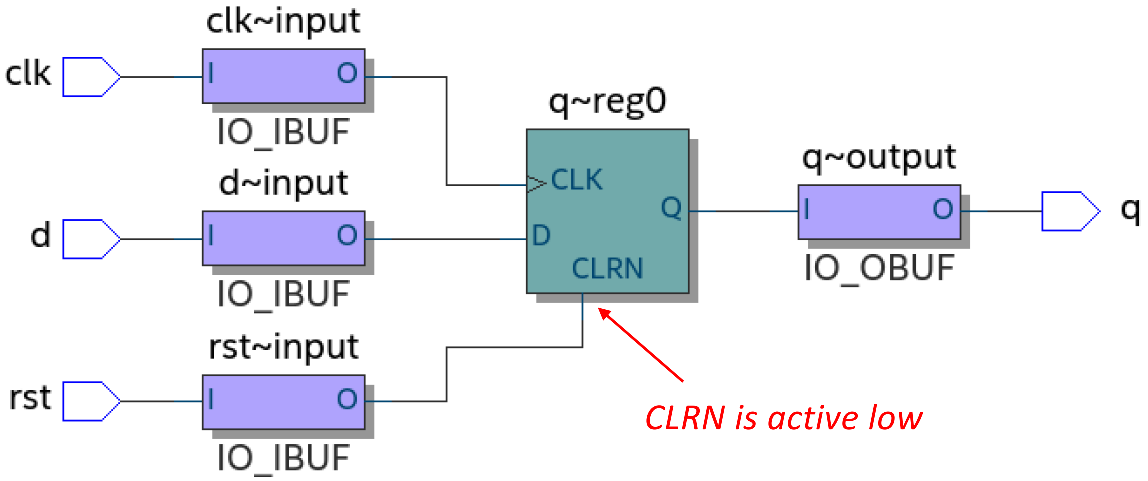
# Review: D flip-flop in Verilog

```
module d_ff(d, clk, q);
  input d, clk;
  output reg q;

  always @(posedge clk)
  begin
    if (clk) q <= d;
  end

  endmodule
```



# Review: D flip-flop with async. reset in Verilog

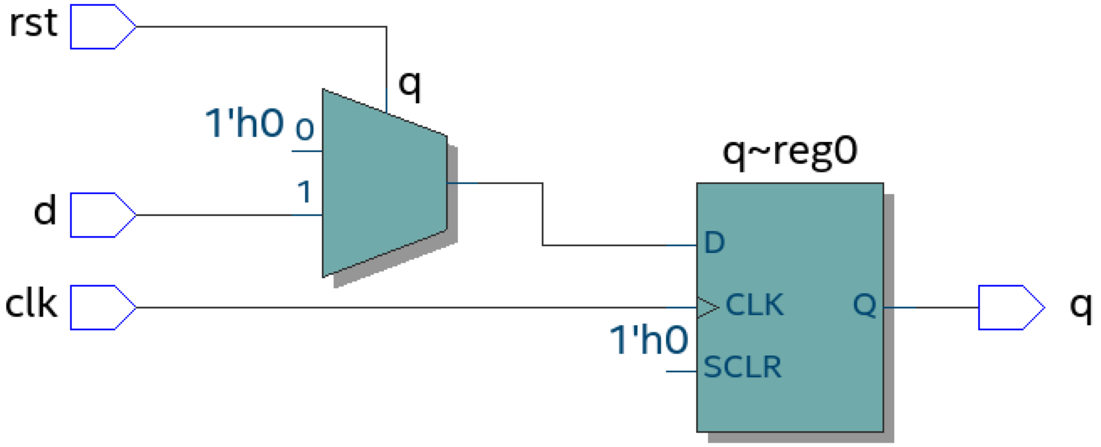
```
module dff_asy (d, rst, clk, q);
  input d, rst, clk;
  output reg q;

  always @(negedge rst, posedge clk)
  begin
    if (~rst)
      q <= 0;
    else
      q <= d;
  end

endmodule
```



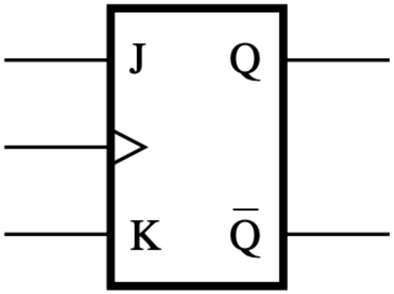
*CLRN is active low*

# Review: D flip-flop with sync. reset in Verilog

```
module dff_syn (d, rst, clk, q);
  input d, rst, clk;
  output reg q;

  always @(posedge clk)
    if (~rst) q <= 0;
    else q <= d;

endmodule
```



# J-K Flip-Flop



| J | K | $Q(t+1) = Q_{next} = Q_{new} = Q_{n+1} = Q^{+}$ |
|---|---|---|
| 0 | 0 | $Q(t)$ |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | $\bar{Q}(t)$ |

| K \ J | 0 | 1 |
|---|---|---|
| 0 | Q | 1 |
| 1 | 0 | Q’ |

→

| K \ J | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 0 | 0 |

| K \ J | 0 | 1 |
|---|---|---|
| 0 | 1 | - |
| 1 | 0 | 0 |

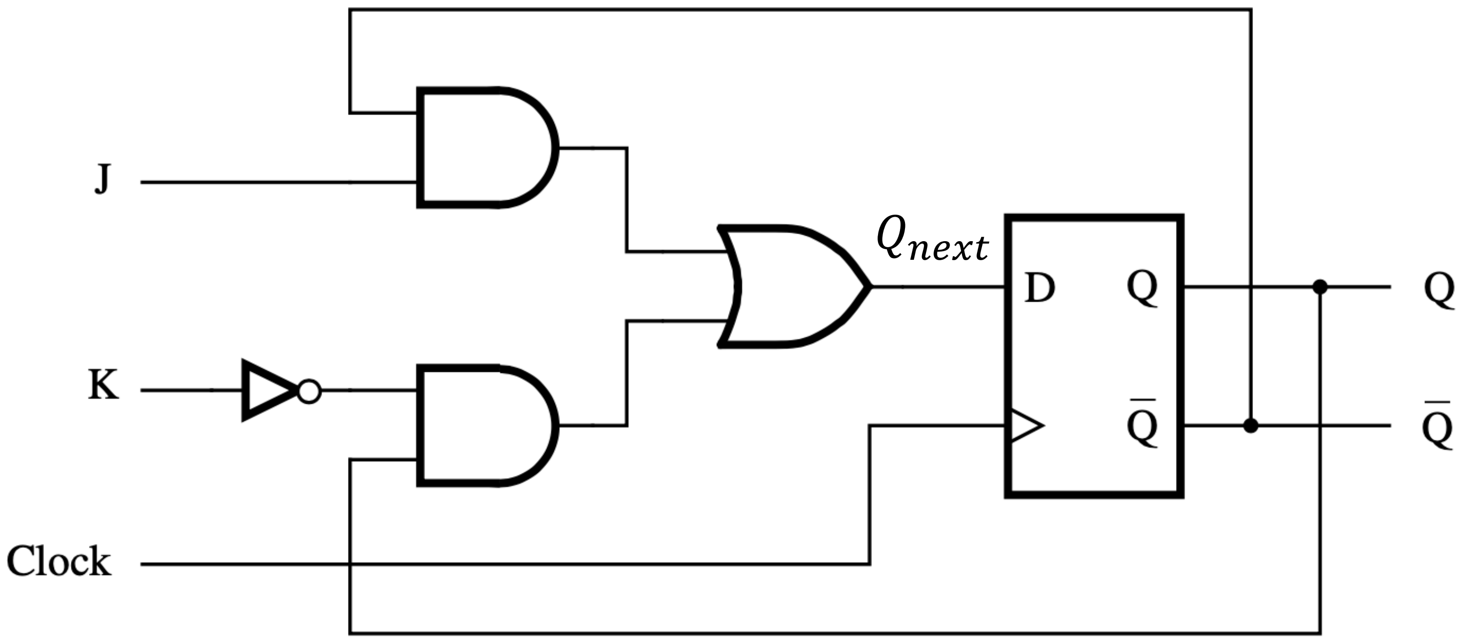
| K \ J | 0 | 1 |
|---|---|---|
| 0 | 0 | - |
| 1 | 0 | 1 |

$$Q_{next} = \underbrace{J \cdot \bar{K}}_{\textit{consensus term}} + Q \cdot \bar{K} + \bar{Q} \cdot J = Q \cdot \bar{K} + \bar{Q} \cdot J$$

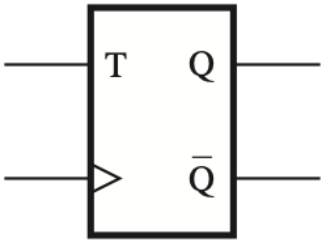
# J-K Flip-Flop schematic

$$Q_{next} = \underbrace{Q \cdot \overline{K} + \bar{Q} \cdot J}$$

*mux with Q as selection and  K’ and J as data*

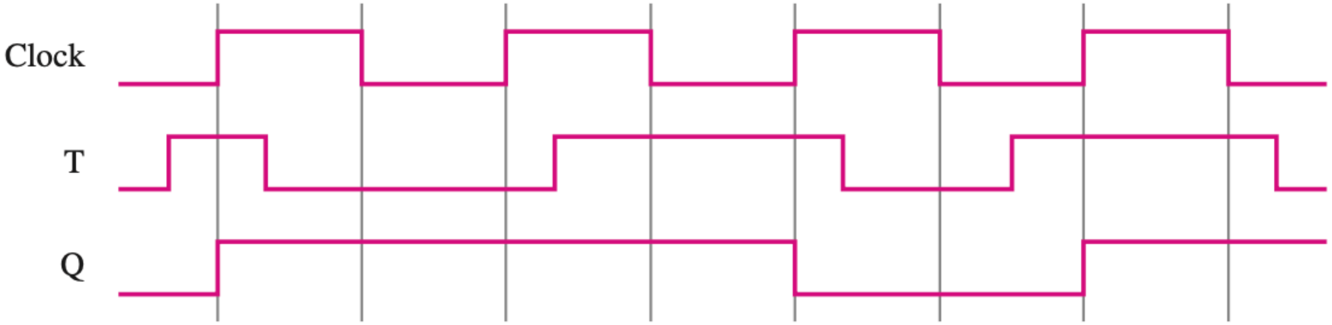


# T Flip-Flop



| T | $Q(t+1) = Q_{next} = Q_{new} = Q_{n+1} = Q^+$ |
|---|---|
| 0 | $Q(t)$ |
| 1 | $\bar{Q}(t)$ |

$$Q_{next} = \bar{T} \cdot Q + T \cdot \bar{Q} = T \oplus Q$$

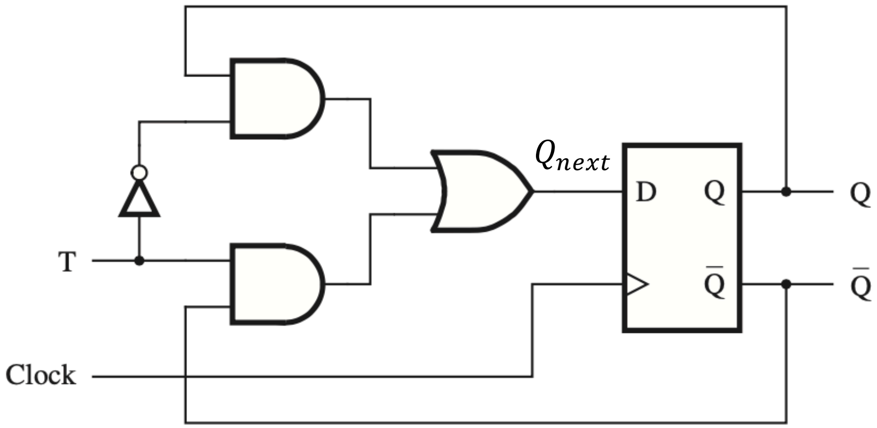


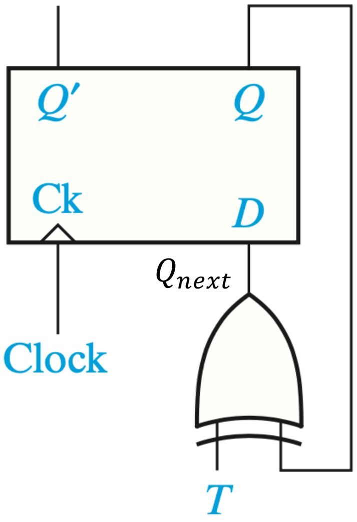
Timing diagram

# T Flip-Flop schematic

$$Q_{next} = \underbrace{\bar{T} \cdot Q + T \cdot \bar{Q}} = T \oplus Q$$

*mux with T as selection*
*and Q’ and Q as data*
*or Q as selection and*
*T’ and T as data*



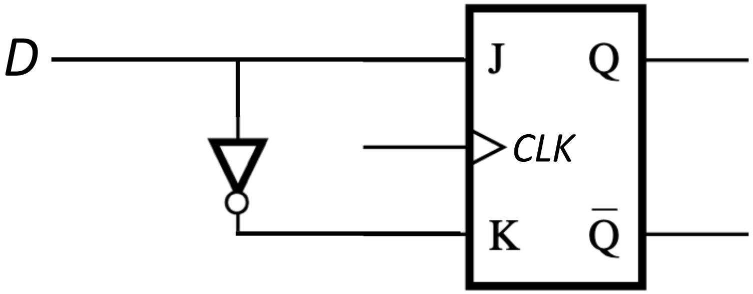


- It is possible to use any flip-flop for building any other flip-flop

# Converting a J-K Flip-Flop into a D Flip-Flop

| J | K | Q | $Q_{next}$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

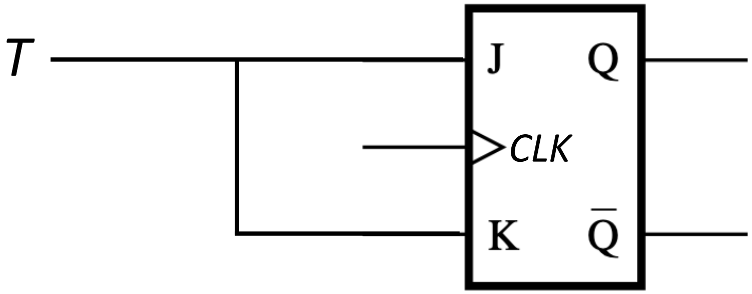
| D | Q | $Q_{next}$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |



# Converting a J-K Flip-Flop into a T Flip-Flop

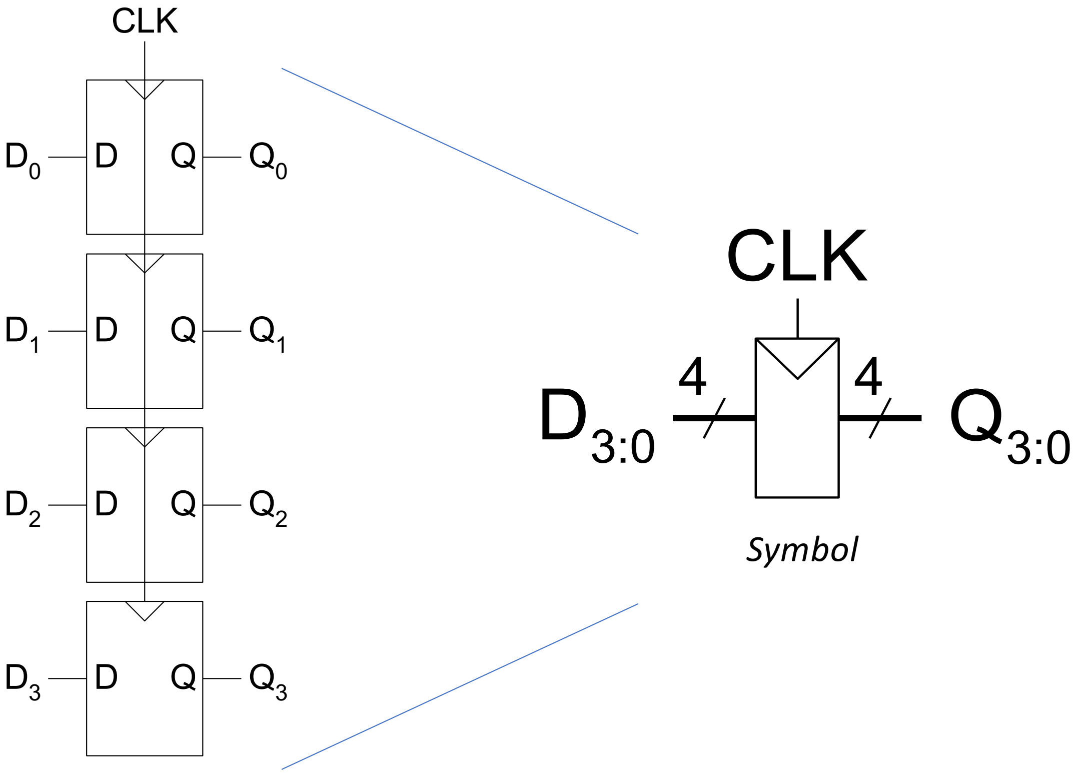
| J | K | $Q_{next}$ |
|---|---|---|
| 0 | 0 | Q |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | Q’ |

| T | $Q_{next}$ |
|---|---|
| 0 | Q |
| 1 | Q’ |



# Register

- An N-bit register is simply a bank of N flip-flops

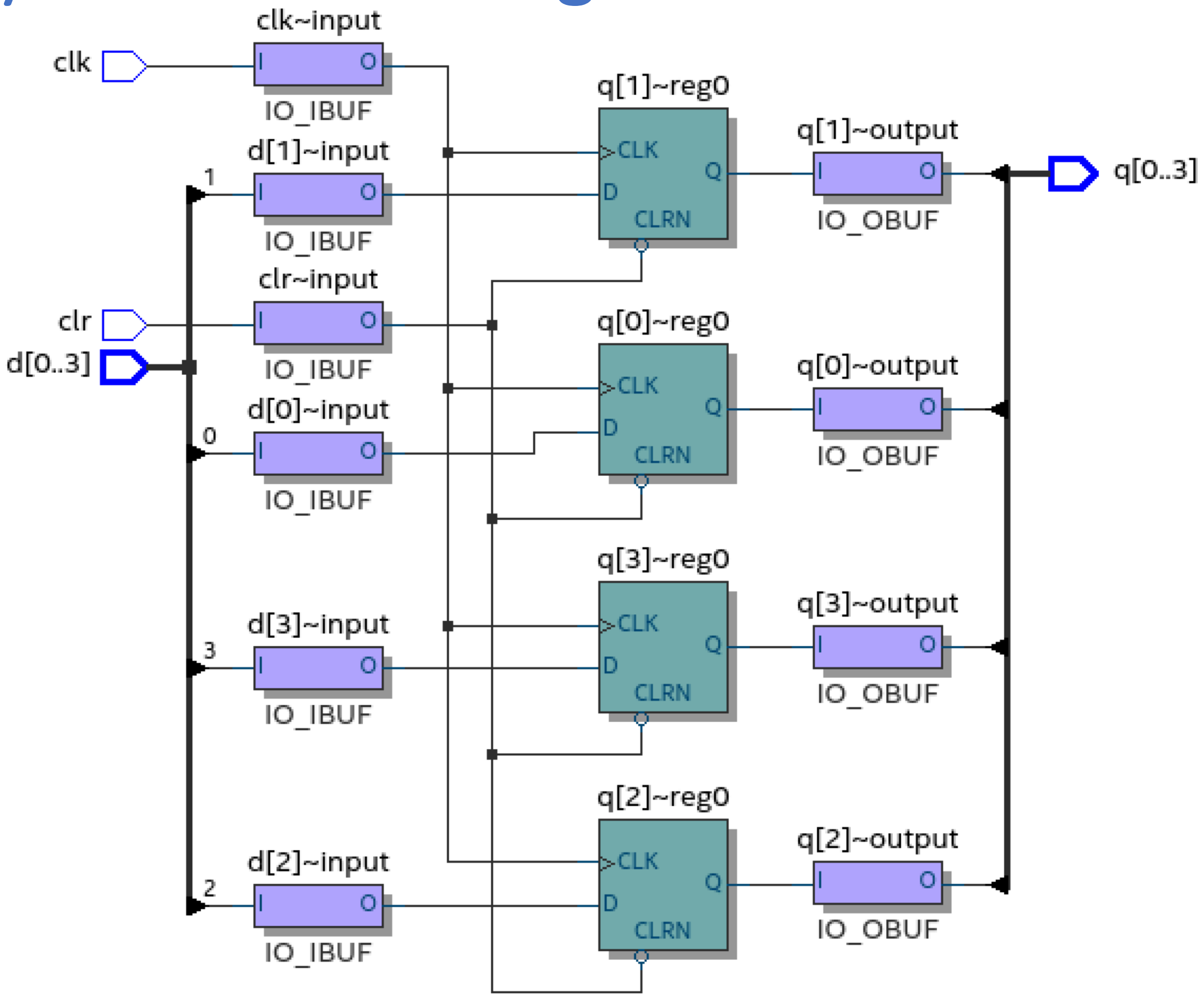


*Symbol*

*Schematic*

# 4-bit register with async. reset: Verilog

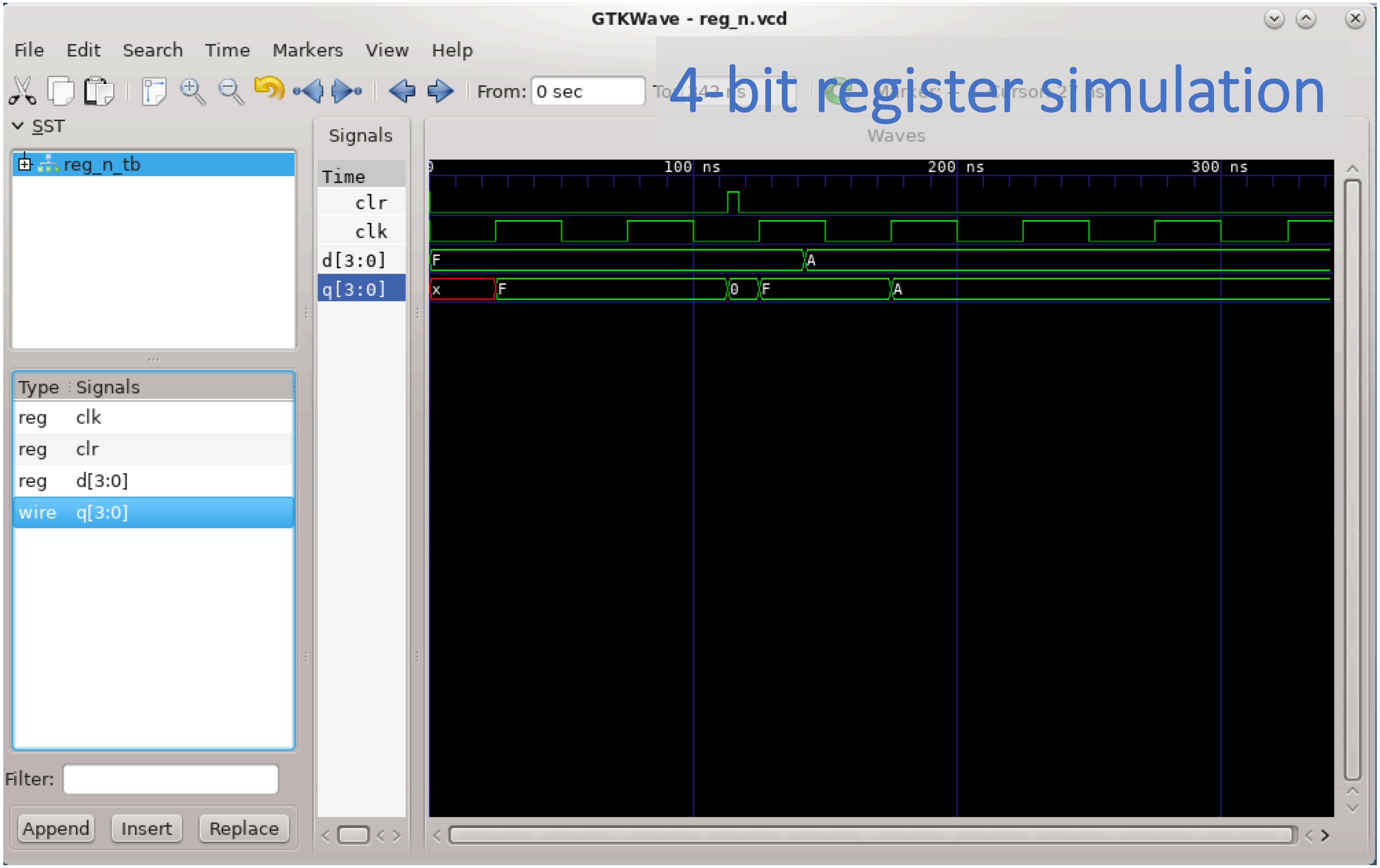
```
// file: reg_n.v
// n-bit register with async. clear

module reg_n(d, clk, clr, q);
  parameter n = 4;
  input [n-1:0] d;
  input clr, clk;
  output reg [n-1:0] q;

  always @(posedge clr, posedge clk)
  begin
    if (clr == 1) q <= 0;
    else q <= d;
  end

  endmodule
```



# 4-bit register testbench

```
// filename: reg_n_tb.v
// verify the functionality of reg_n.v

// set time tick units and precision to ns
‘timescale 1ns / 1ns

module reg_n_tb;

  // UUT inputs
  reg [3:0] d;
  reg clk;
  reg clr;
  // UUT outputs
  wire [3:0] q;

  // parameters
  parameter T = 50; // clock period

  // instantiate the unit under test (UUT)
  reg_n uut(
    .d(d),
    .clk(clk),
    .clr(clr),
    .q(q)
  );

  // generate clock
  // 50 ns clock running forever
  always
  begin
    clk = 1’b0;
    #(T/2);
    clk = 1’b1;
    #(T/2);
  end

  // initialize inputs
  initial
  begin
    d   = 4’hf;
    clr = 0;
  end

  // set time format to be ns
  initial $timeformat(-9, 1, "ns", 10);

  // output the simulation in graphical format
  initial begin
    $dumpfile("reg_n.vcd");
    $dumpvars(0, reg_n_tb);
  end

  // generate test patterns
  initial begin
    // test async. clear
    #113;
    clr = 1;
    #4;
    clr = 0;
    // test the register sample a new data value
    #25;
    d = 4’ha;
    #200;
    $finish;
  end

  // output the simulation in textual format
  initial begin
    $display("\ntime clr d q");
    $display("----------------");
    $monitor("%4d  %1b %h %h", $time, clr, d, q);
  end

endmodule
```

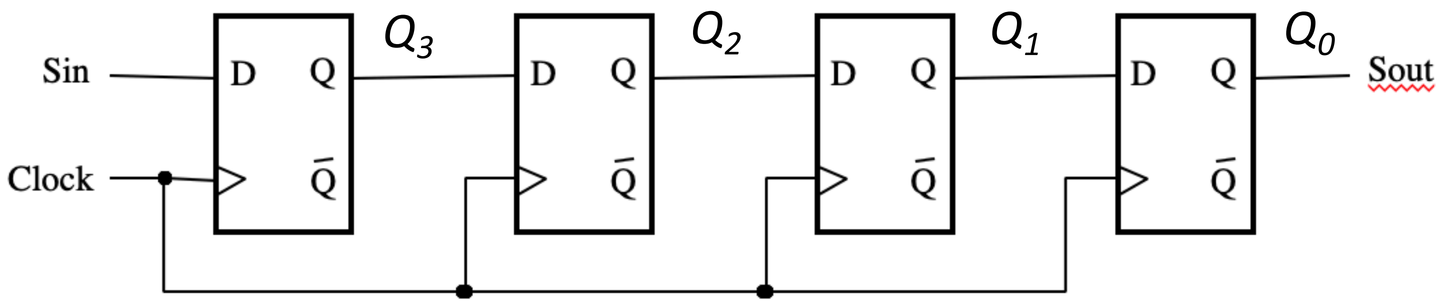
# 4-bit register simulation



| Type | Signals |
|---|---|
| reg | clk |
| reg | clr |
| reg | d[3:0] |
| wire | q[3:0] |

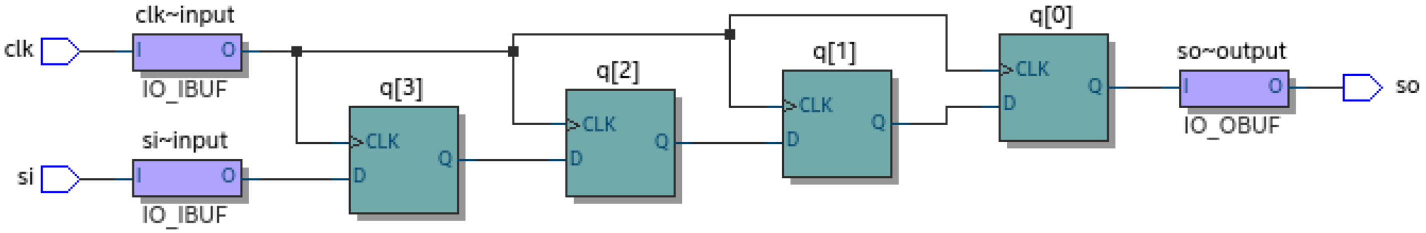
# Shift-register

*Example: 4-bit shift register*



# Shift Register in Verilog

```
// file: sreg_n.v
// n-bit shift register

module sreg_n(si, clk, so);
  parameter n = 4;
  input si, clk;
  output so;

  reg [n-1:0] q;
  integer k;

  always @(posedge clk)
  begin
    for (k=0; k<n-1; k=k+1) q[k] <= q[k+1];
    q[n-1] <= si;
  end

  assign so = q[0];

  endmodule
```

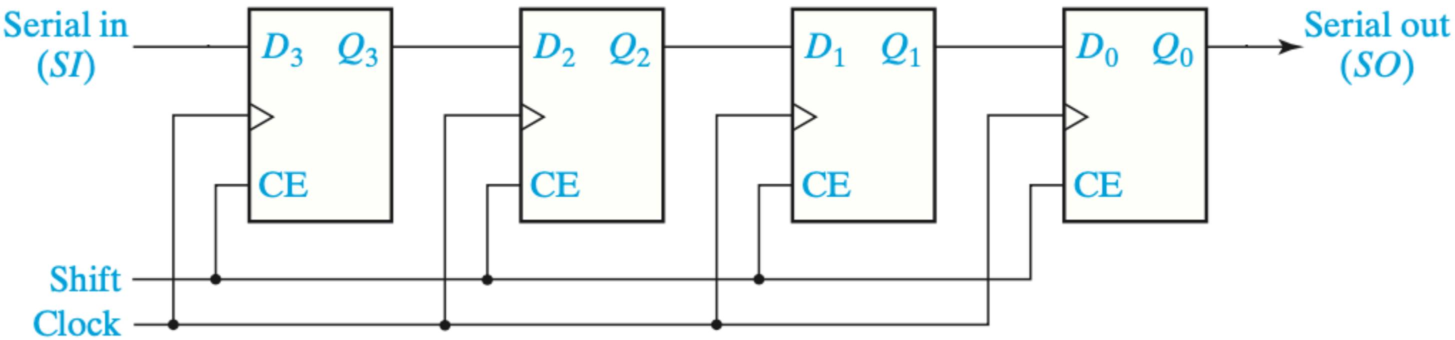


# Shift Register (with shift enable)

*(Example: 4-bit)*

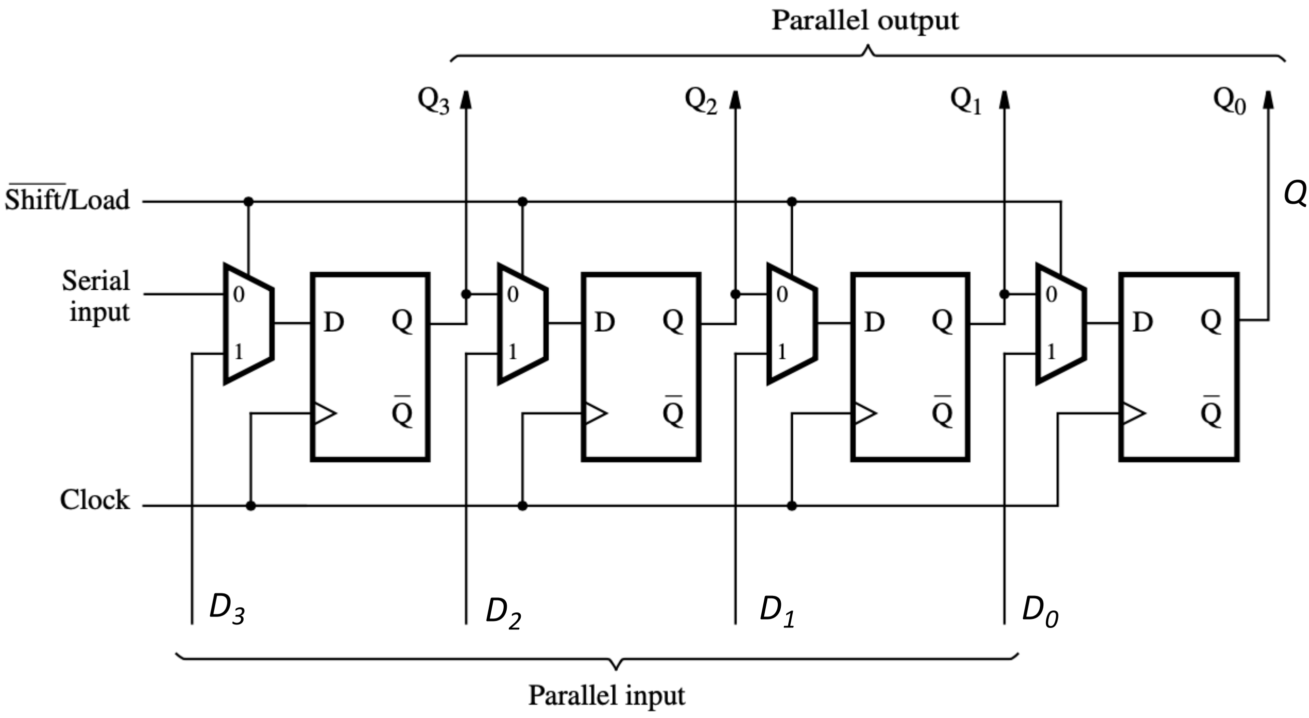
*CE = clock enable*

Right-Shift Register



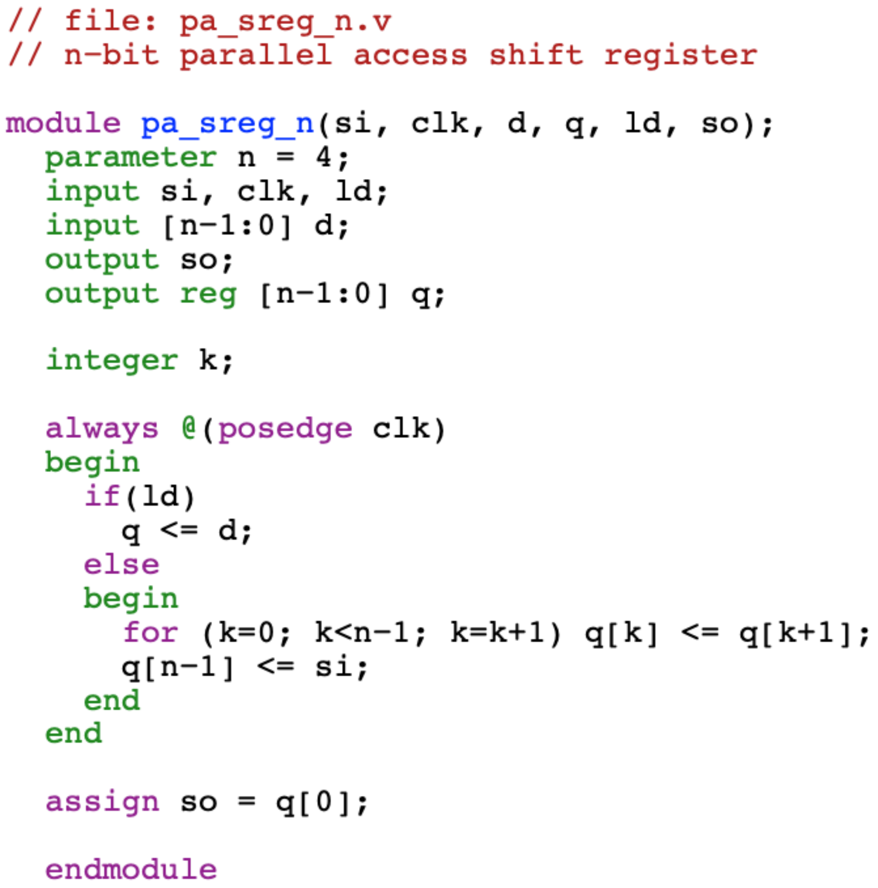
# Parallel Access Shift Register

*(Example: 4-bit)*



*$Q_0$ = Serial output*

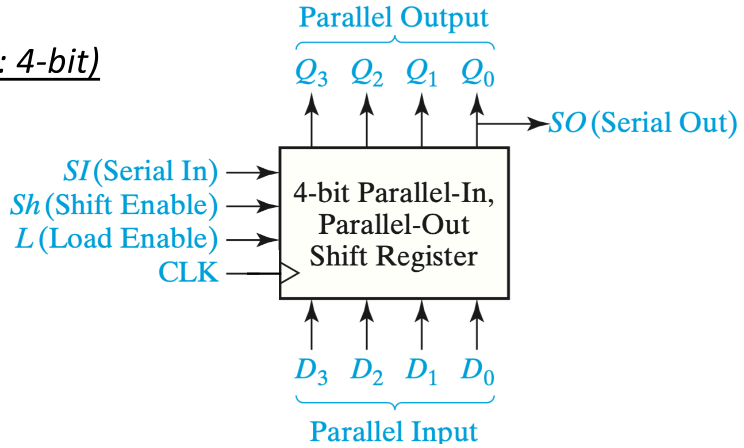
# Parallel Access Shift Register in Verilog



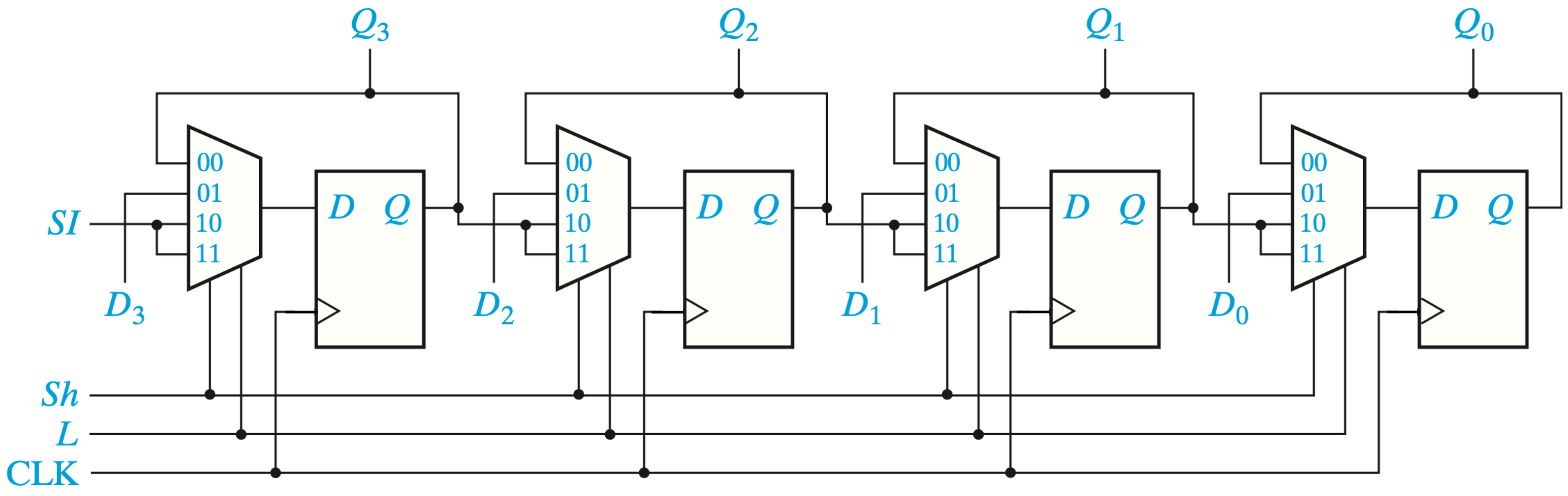
```verilog
// file: pa_sreg_n.v
// n-bit parallel access shift register

module pa_sreg_n(si, clk, d, q, ld, so);
  parameter n = 4;
  input si, clk, ld;
  input [n-1:0] d;
  output so;
  output reg [n-1:0] q;

  integer k;

  always @(posedge clk)
  begin
    if(ld)
      q <= d;
    else
    begin
      for (k=0; k<n-1; k=k+1) q[k] <= q[k+1];
      q[n-1] <= si;
    end
  end

  assign so = q[0];

  endmodule
```

# Parallel Access Shift Register (with shift enable)

*(Example: 4-bit)*



*Shift Enable has priority on Load Enable*

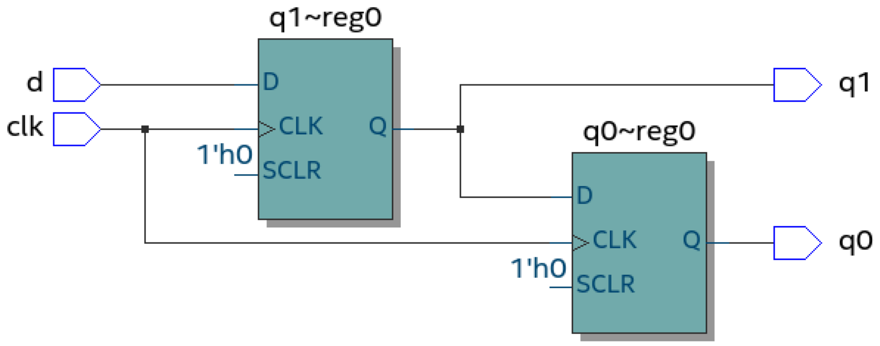


# Next Time

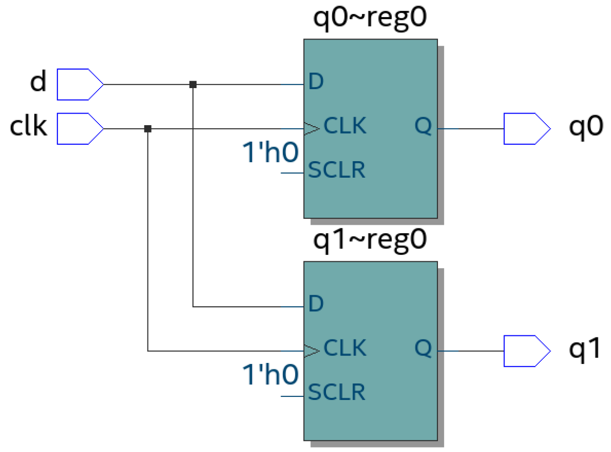
- Counters
  - Asynchronous (don’t use them!)
  - Synchronous
- Finite State Machines
  - Mealy
  - Moore

# When writing Verilog think Hardware !

```
// file: pipe.v
// flip-flops pipelined
module pipe(d, clk, q1, q0);
  input d, clk;
  output reg q1, q0;

  always @(posedge clk)
  begin
    if (clk)
      q1 <= d;  //nonblocking
      q0 <= q1; //nonblocking
  end

endmodule
```



```
// file: red.v
// redundant flip flop
module red(d, clk, q1, q0);
  input d, clk;
  output reg q1, q0;

  always @(posedge clk)
  begin
    if (clk)
      q1 <= d;  //nonblocking
      q0 <= d;  //nonblocking
  end

endmodule
```



# S.L. Building Blocks (continued)

CPEN 230 – Introduction to Digital Logic

# Review

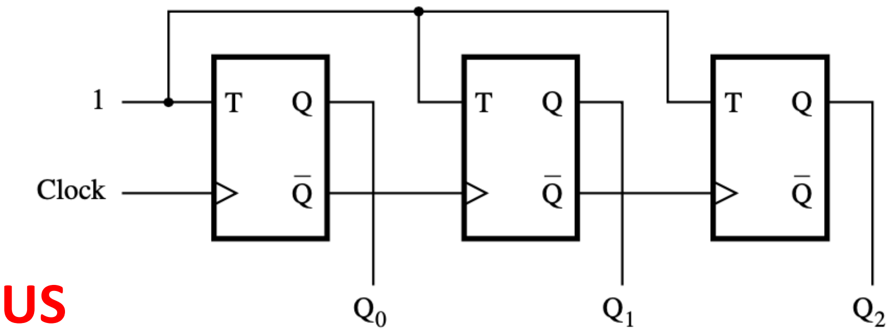
- D latch
- D flip-flop
- J-K flip-flop
- T flip-flop
- Registers
- Shift Registers

# Counters

- Two categories of counters
    - Asynchronous (to be avoided at all costs)
    - Synchronous
        - binaries counters (T flip-flop based and D-flip-flop based)
        - shift register counters (ring counter and Johnson Counter) --> Fast Counters

# Asynchronous modulo-$2^N$ upward counter

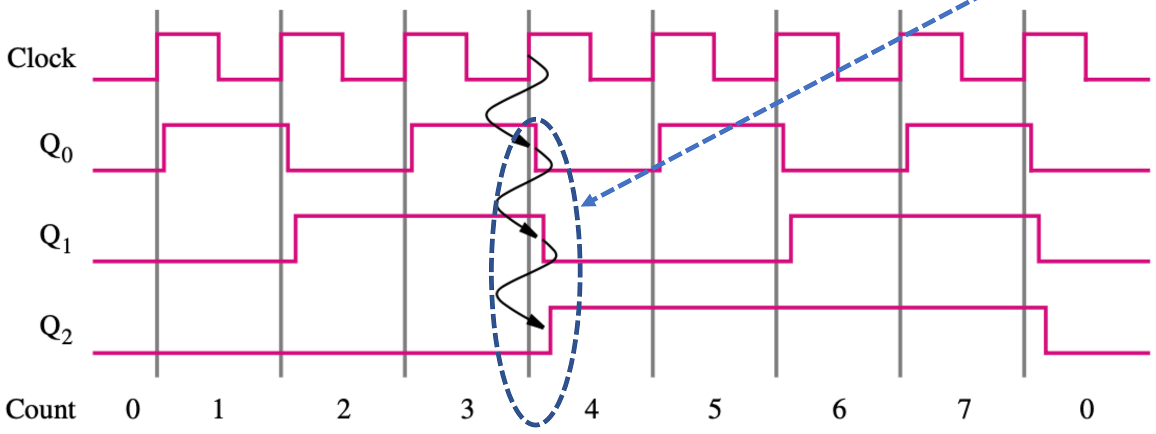
*Example: N=3*



(a) Circuit

**AVOID ASYNCHRONOUS COUNTERS !**

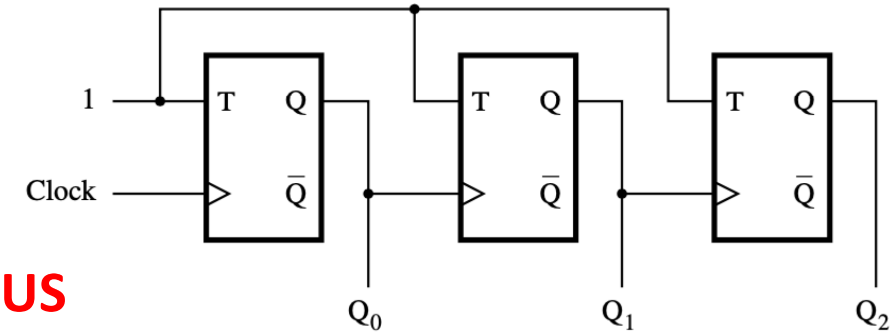
*The delay is not the same.*
*It increases with each flip-flop*
*we go through.*



(b) Timing diagram

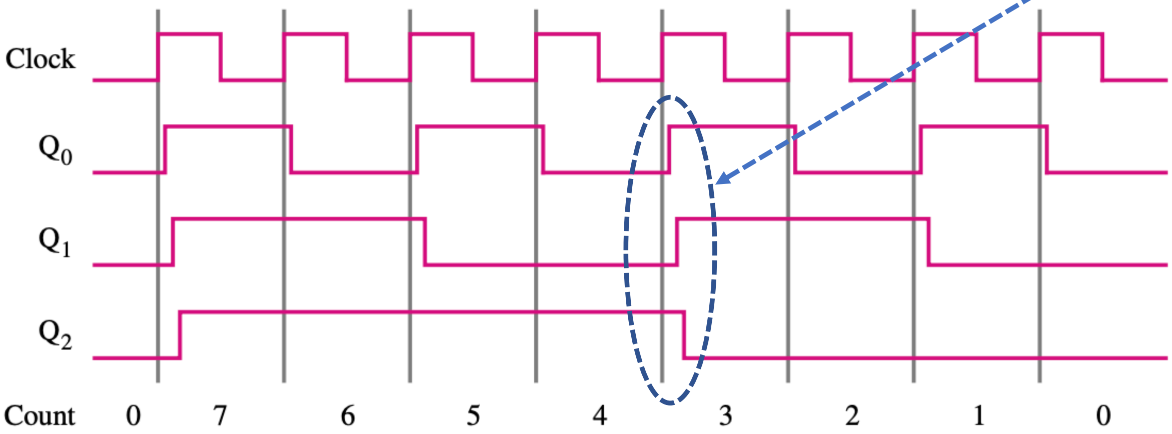
# Asynchronous modulo-$2^N$ downward counter

*Example: N=3*



(a) Circuit

**AVOID ASYNCHRONOUS COUNTERS !**



(b) Timing diagram

*The delay is not the same. It Increases with each flip-flop we go through.*

# Synchronous counters

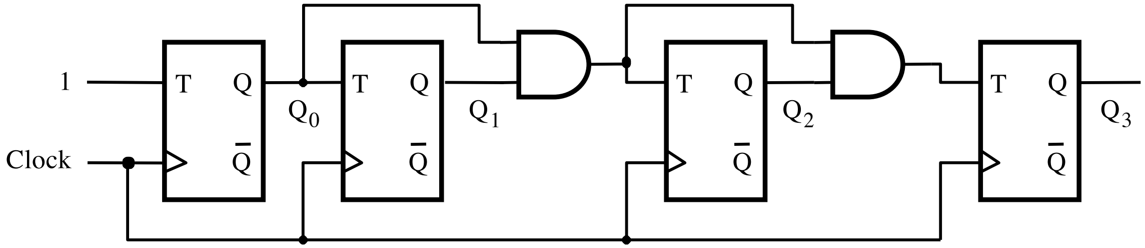
- Synchronous counters are built by clocking all flip-flops with a single clocking source (this makes them more reliable than asynchronous counters)

# Binary counter (using T flip-flops)

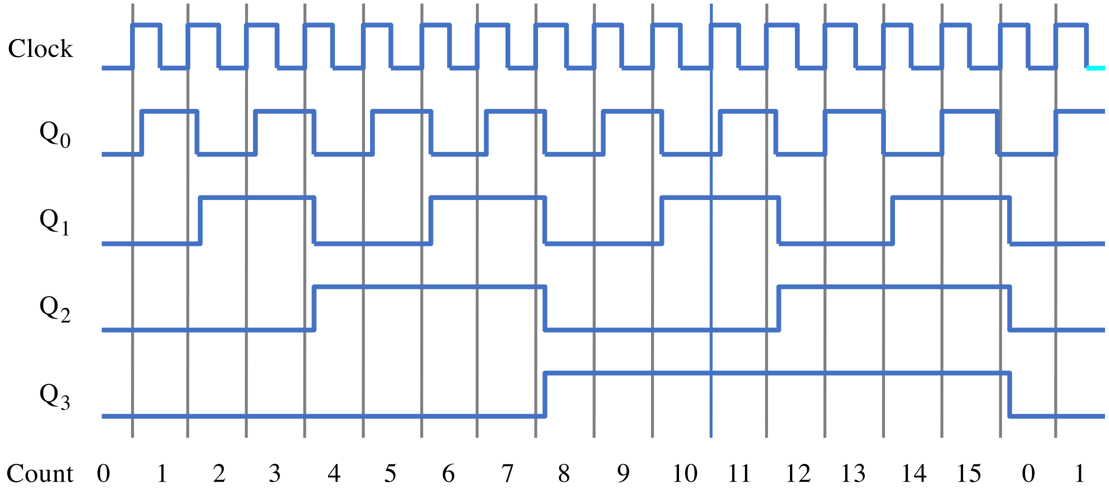
| count | $Q_2$ | $Q_1$ | $Q_0$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 |
| 2 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 |
| 4 | 1 | 0 | 0 |
| 5 | 1 | 0 | 1 |
| 6 | 1 | 1 | 0 |
| 7 | 1 | 1 | 1 |

The operation of the counter is based on the observation that the state of the flip-flop in stage *i flips* only if all preceding flip-flops are in the state Q = 1.

$T_0 = 1$
$T_1 = Q_0$
$T_2 = Q_0 Q_1$
…
$T_{n-1} = Q_0 Q_1 \cdots Q_{n-2}$

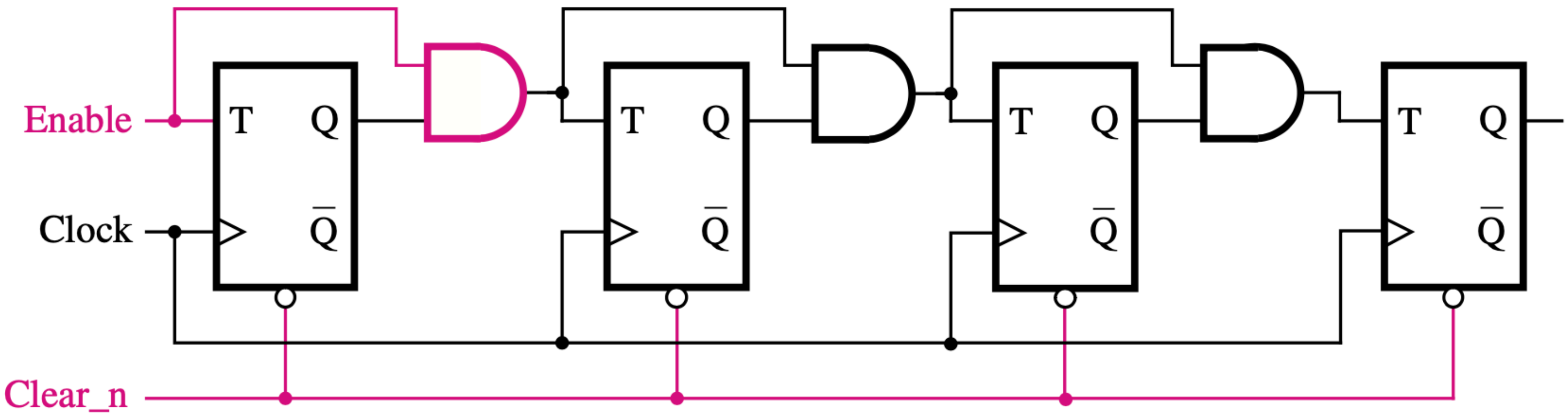


(a) Circuit for 4-bit binary counter



(b) Timing diagram

# Adding Enable and Clear Capability

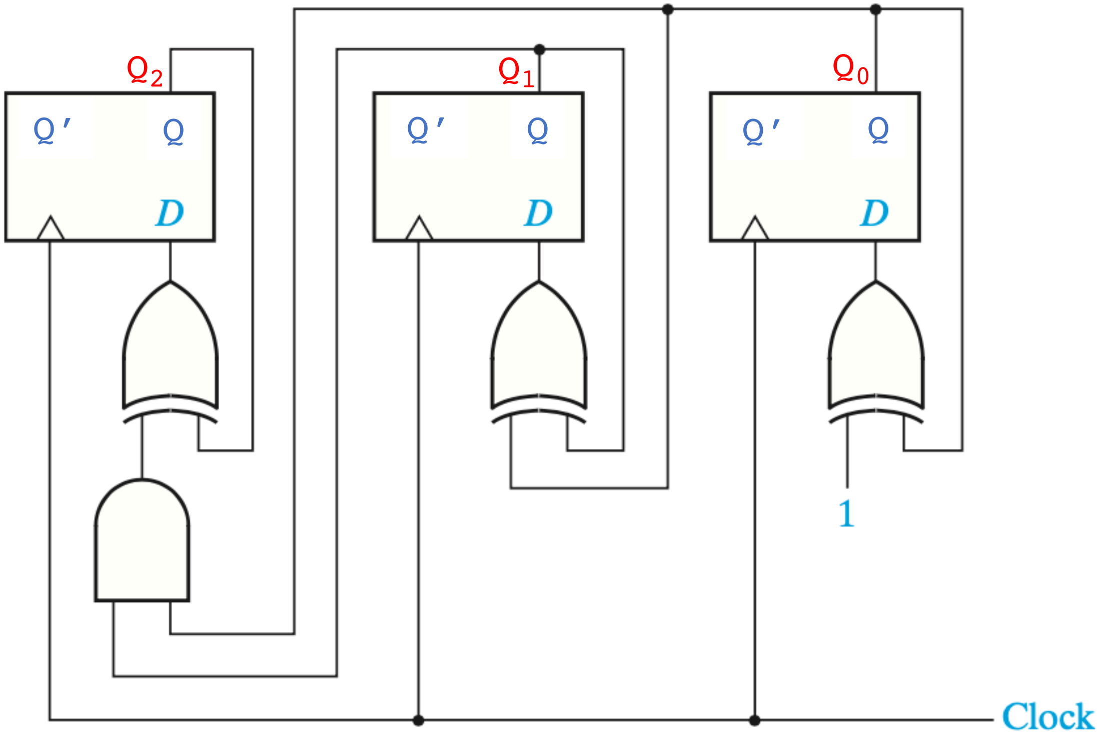


# Binary counter (using D flip-flops)

The operation of the counter is based on the observation that the state of the flip-flop in stage *i flip*s only if all preceding flip-flops are in the state Q = 1.

| Clock cycle | $Q_2\,Q_1\,Q_0$ | $(Q_2\,Q_1\,Q_0)^{next}$ |
|---|---|---|
| 0 | 0 0 0 | 0 0 1 |
| 1 | 0 0 1 | 0 1 0 |
| 2 | 0 1 0 | 0 1 1 |
| 3 | 0 1 1 | 1 0 0 |
| 4 | 1 0 0 | 1 0 1 |
| 5 | 1 0 1 | 1 1 0 |
| 6 | 1 1 0 | 1 1 1 |
| 7 | 1 1 1 | 0 0 0 |

$$D_0 = \overline{Q_0} = Q_0 \oplus 1$$
$$D_1 = Q_1 \oplus Q_0$$
$$D_2 = Q_2 \oplus Q_1 Q_0$$
$$\dots$$
$$D_{n-1} = Q_{n-1} \oplus Q_{n-2} \cdots Q_1 Q_0$$



# Adding Enable and Clear capability

$D_0 = Q_0 \oplus Enable$

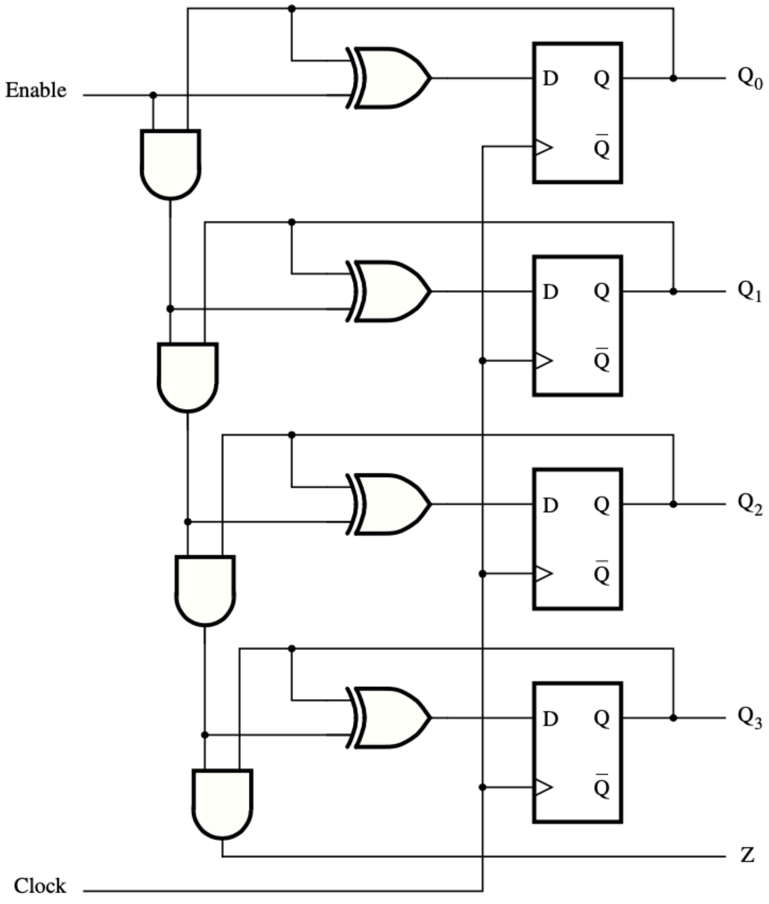
$D_1 = Q_1 \oplus Q_0 \cdot Enable$

$D_2 = Q_2 \oplus Q_1 \cdot Q_0 \cdot Enable$

$D_3 = Q_3 \oplus Q_2 \cdot Q_1 \cdot Q_0 \cdot Enable$

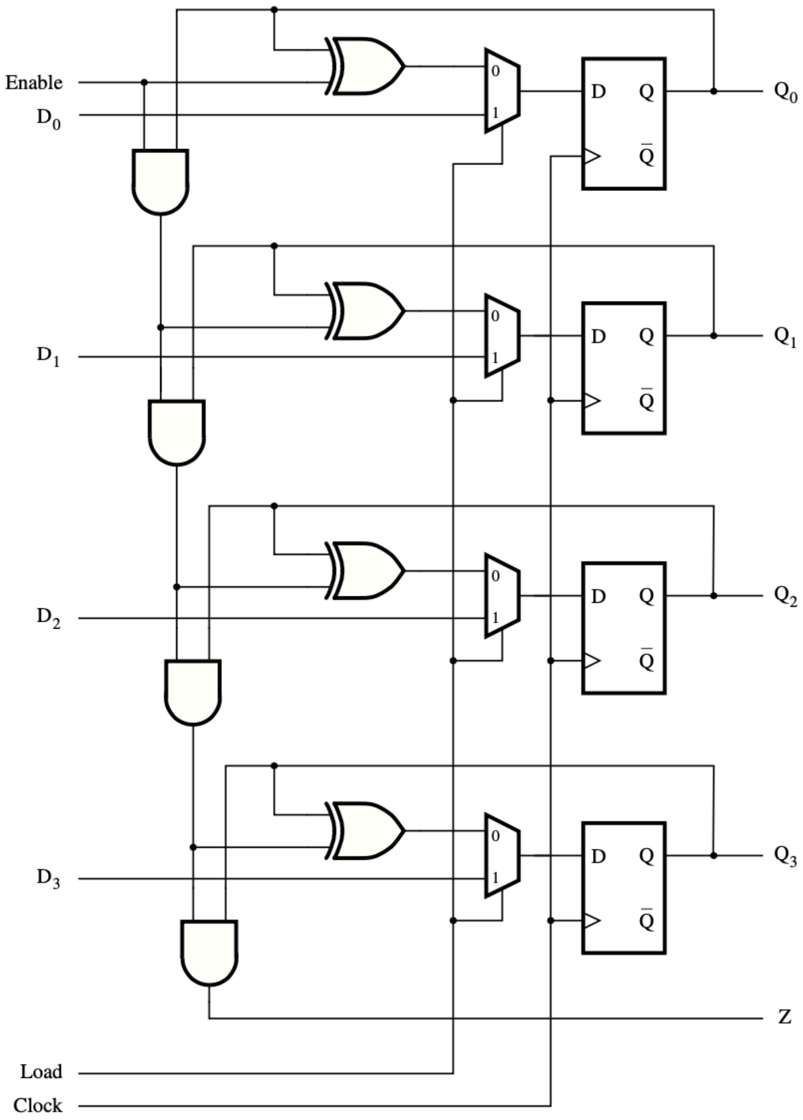
…

If Enable = 0 the state of the flip-flop in stage *i does not change* $(Q_i \oplus 0 = Q_i)$

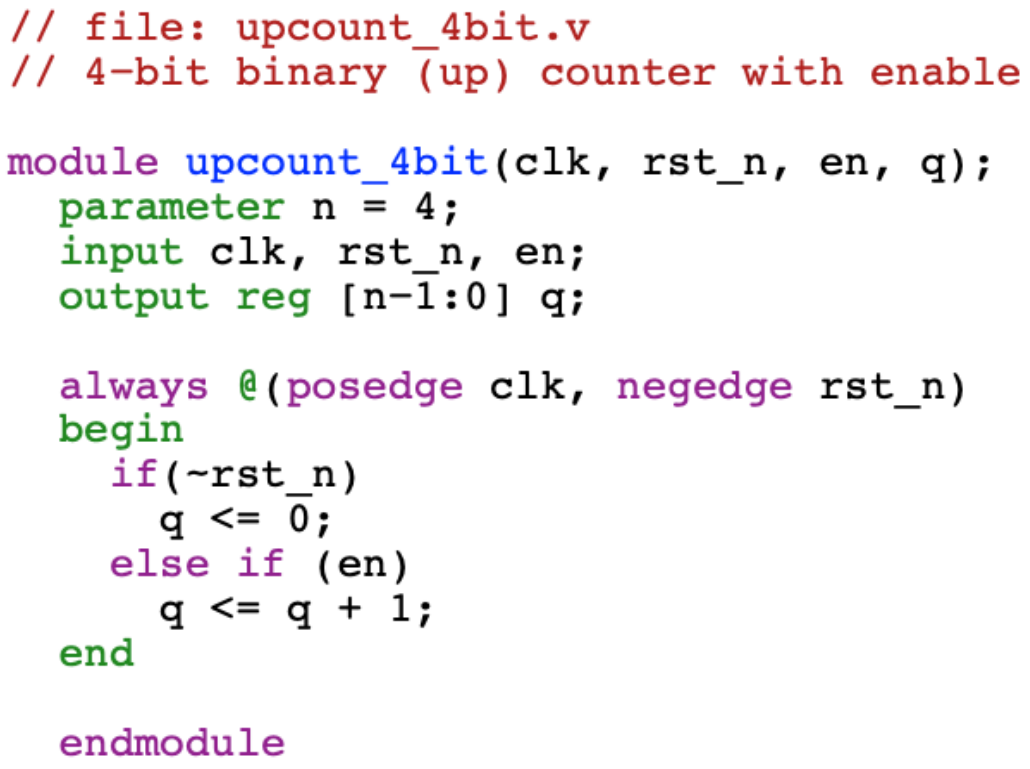


*In case we want to cascade another counter (Z serves as the Enable for the next counter )*

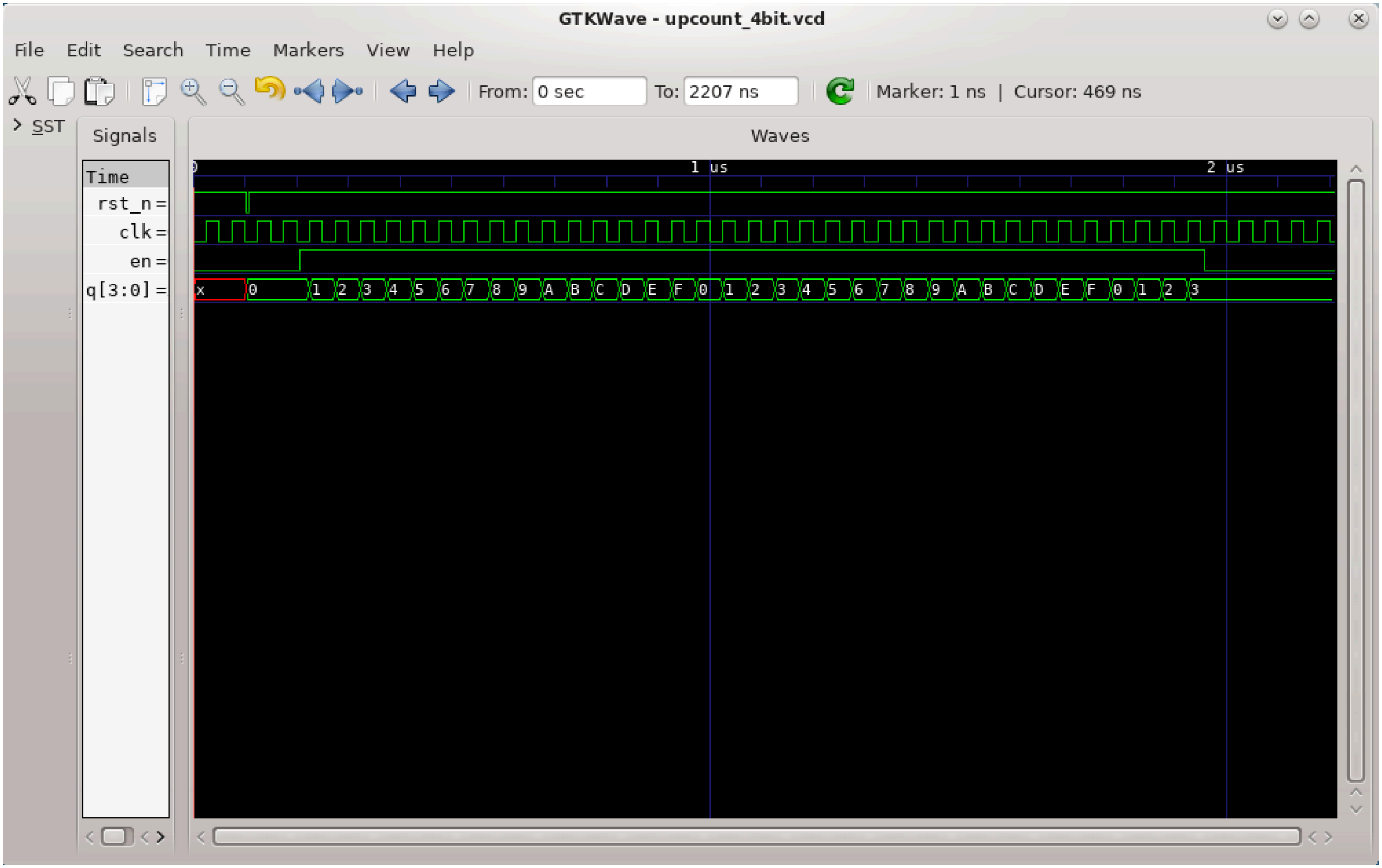
# Adding parallel-load capability



# “Counting” in Verilog



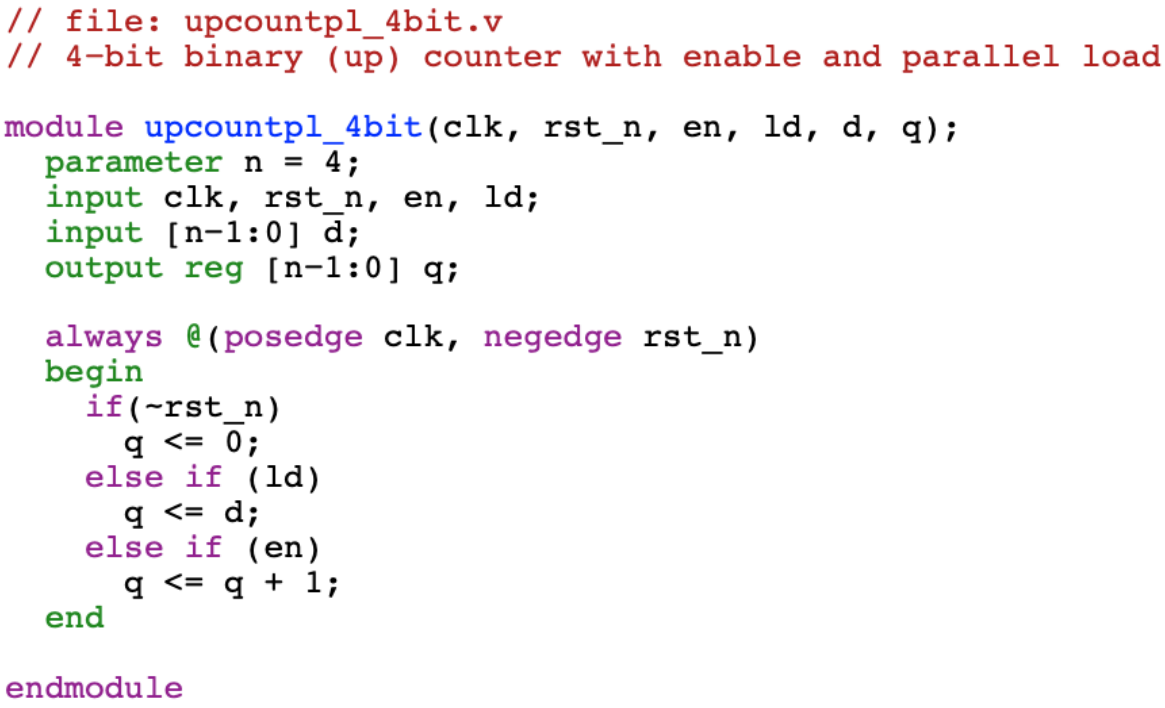
```
// file: upcount_4bit.v
// 4-bit binary (up) counter with enable

module upcount_4bit(clk, rst_n, en, q);
  parameter n = 4;
  input clk, rst_n, en;
  output reg [n-1:0] q;

  always @(posedge clk, negedge rst_n)
  begin
    if(~rst_n)
      q <= 0;
    else if (en)
      q <= q + 1;
  end

  endmodule
```



GTKWave - upcount_4bit.vcd

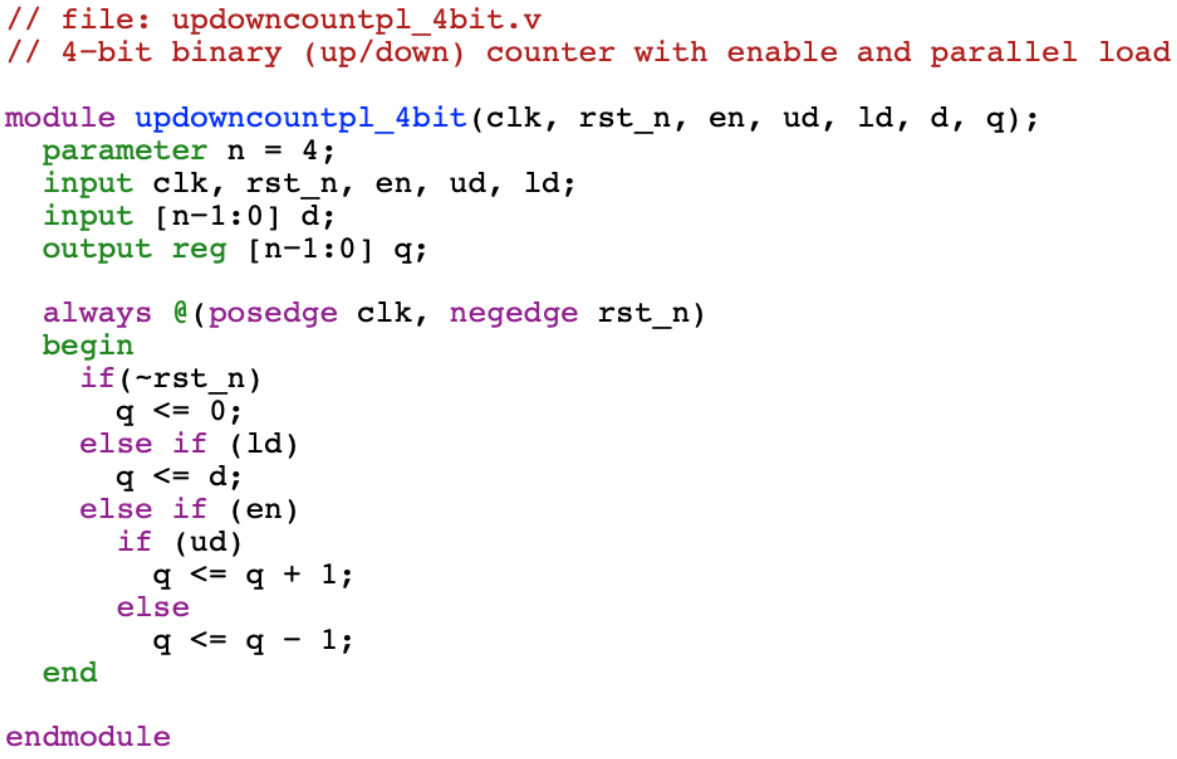
From: 0 sec  To: 2207 ns  Marker: 1 ns | Cursor: 469 ns

# Example: adding parallel-load capability



```
// file: upcountpl_4bit.v
// 4-bit binary (up) counter with enable and parallel load

module upcountpl_4bit(clk, rst_n, en, ld, d, q);
  parameter n = 4;
  input clk, rst_n, en, ld;
  input [n-1:0] d;
  output reg [n-1:0] q;

  always @(posedge clk, negedge rst_n)
  begin
    if(~rst_n)
      q <= 0;
    else if (ld)
      q <= d;
    else if (en)
      q <= q + 1;
  end

endmodule
```

# Example: adding down-count capability

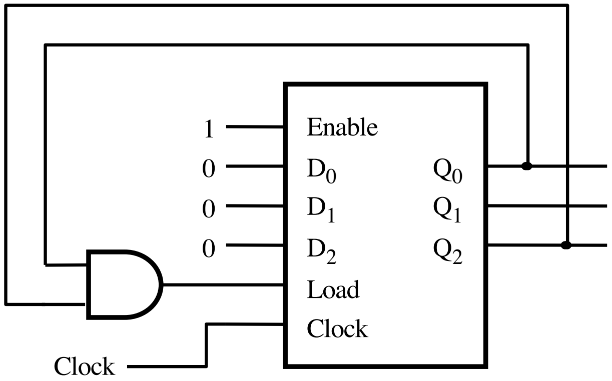


```
// file: updowncountpl_4bit.v
// 4-bit binary (up/down) counter with enable and parallel load

module updowncountpl_4bit(clk, rst_n, en, ud, ld, d, q);
  parameter n = 4;
  input clk, rst_n, en, ud, ld;
  input [n-1:0] d;
  output reg [n-1:0] q;

  always @(posedge clk, negedge rst_n)
  begin
    if(~rst_n)
      q <= 0;
    else if (ld)
      q <= d;
    else if (en)
      if (ud)
        q <= q + 1;
      else
        q <= q - 1;
  end

endmodule
```

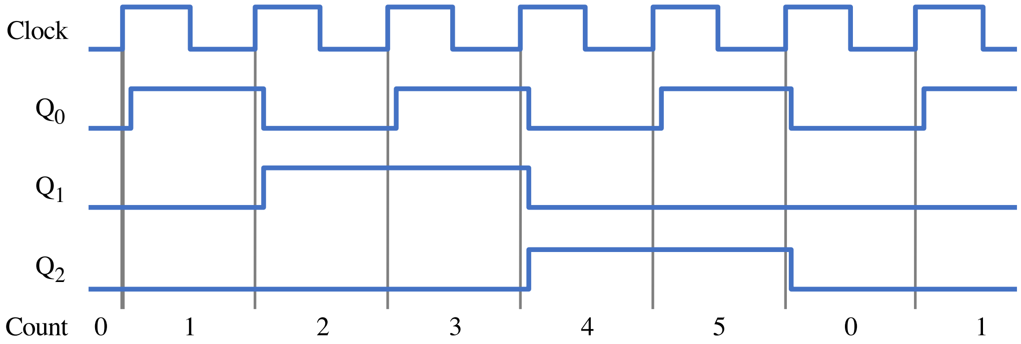
# module-M counter

Not all the time we need to count for $2^N$

*Example: module-6 counter*

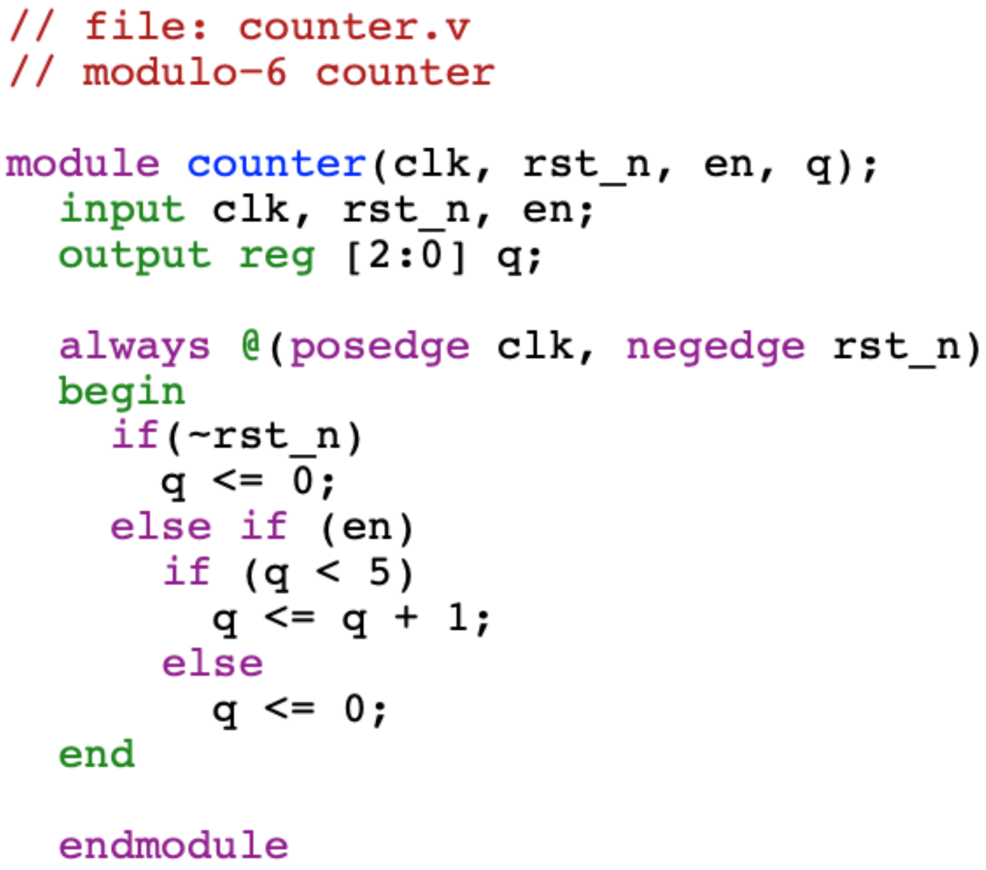


(a) Circuit

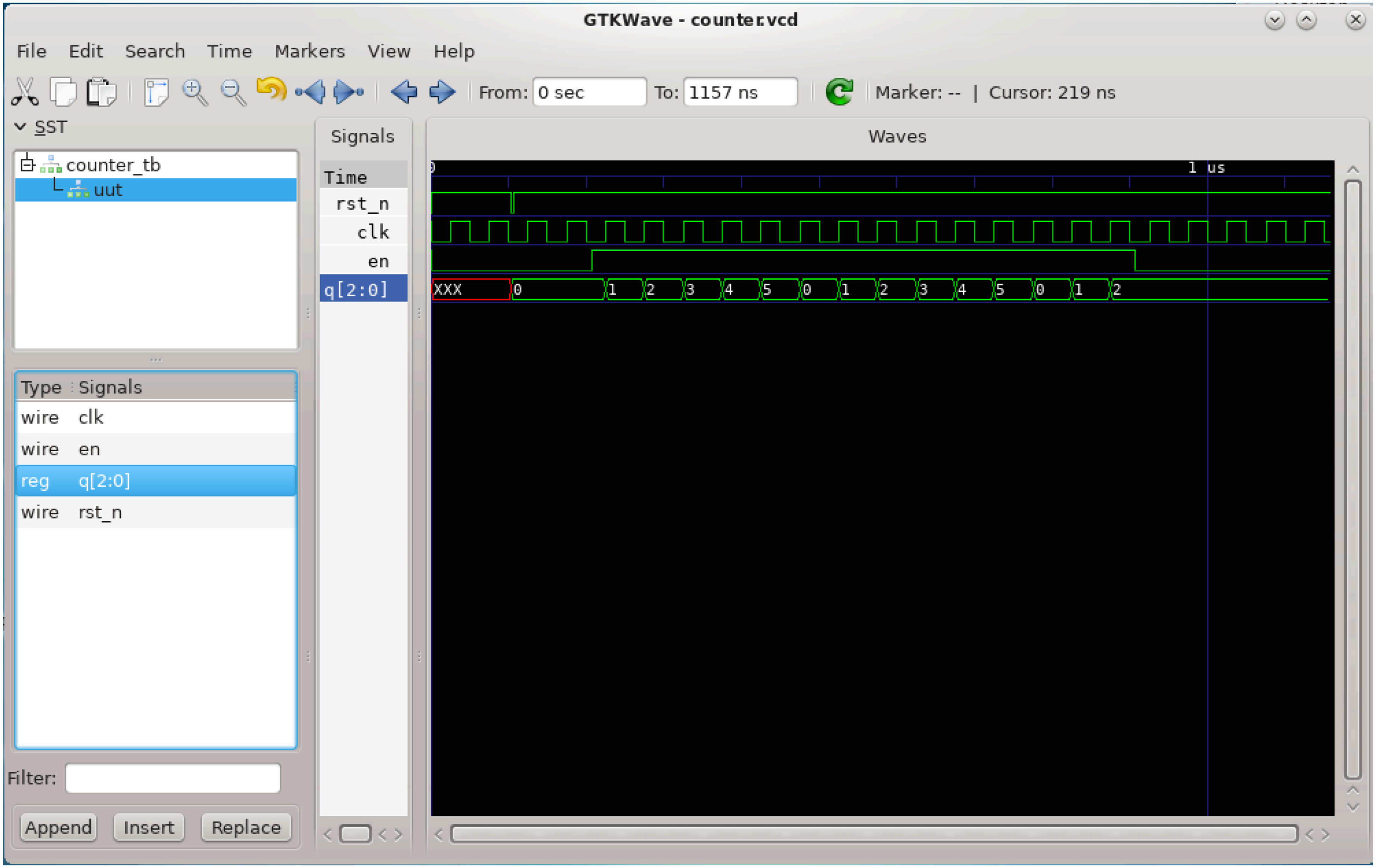


(b) Timing diagram

# Example: modulo-6 counter



```
// file: counter.v
// modulo-6 counter

module counter(clk, rst_n, en, q);
  input clk, rst_n, en;
  output reg [2:0] q;

  always @(posedge clk, negedge rst_n)
  begin
    if(~rst_n)
      q <= 0;
    else if (en)
      if (q < 5)
        q <= q + 1;
      else
        q <= 0;
  end

  endmodule
```



From: 0 sec To: 1157 ns

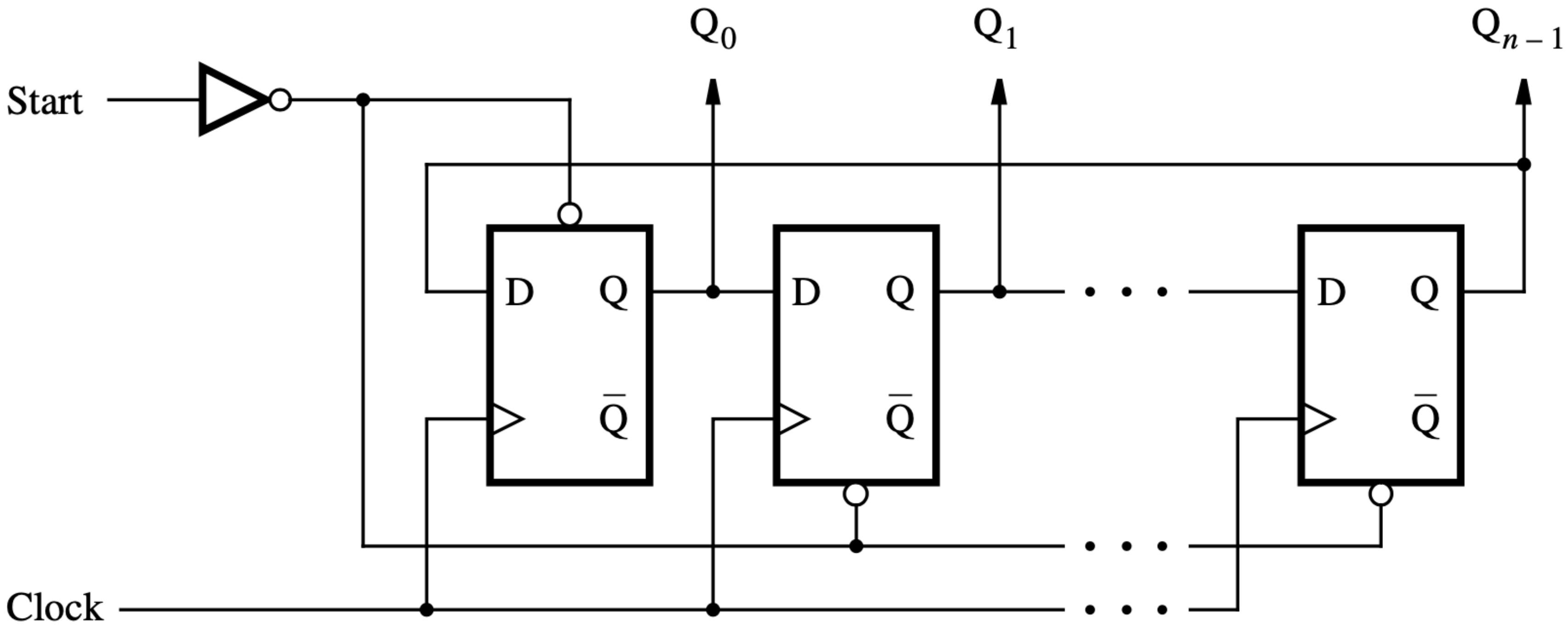
Marker: -- | Cursor: 219 ns

| Type | Signals |
|---|---|
| wire | clk |
| wire | en |
| reg | q[2:0] |
| wire | rst_n |

# Ring counter

An n-bit counter of this type generates a counting sequence of length n

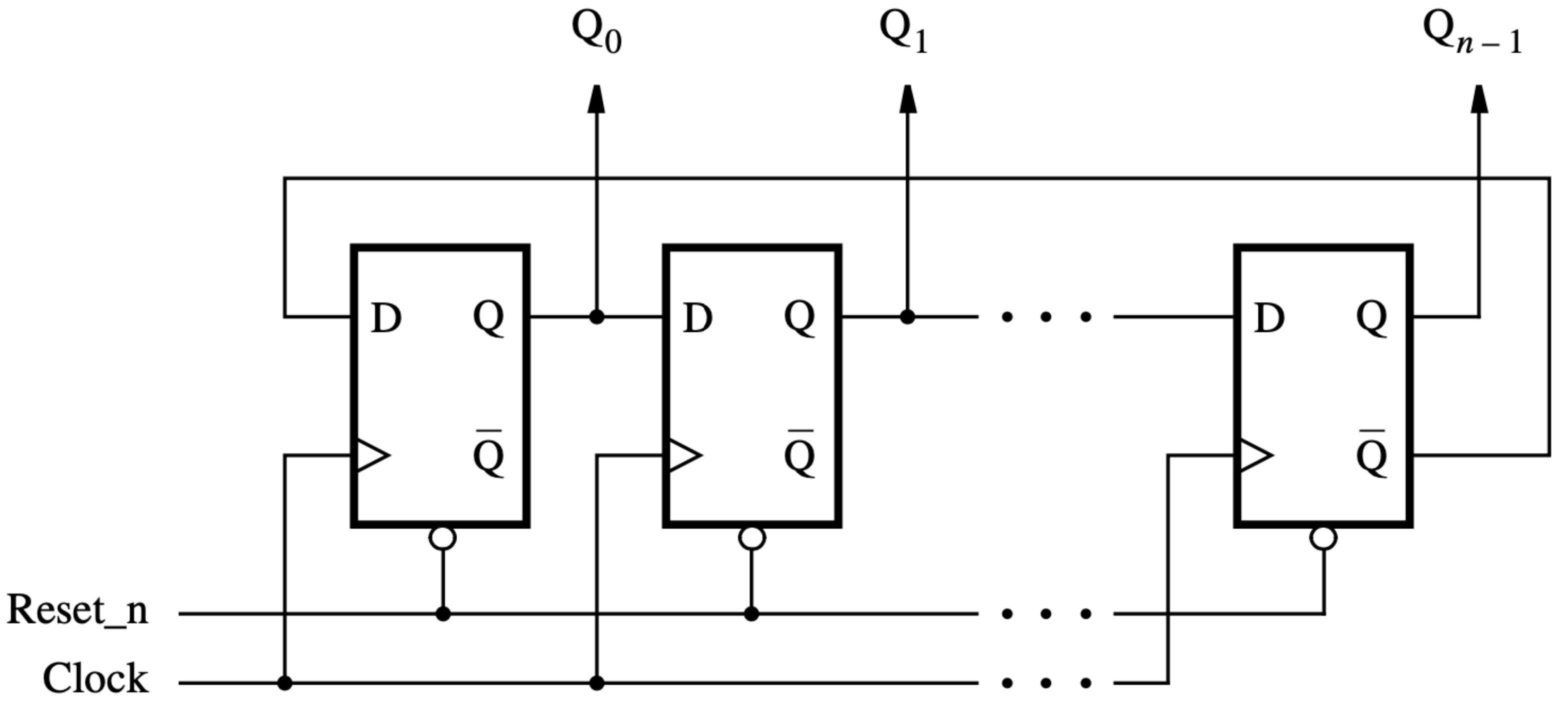
For example a 4-bit counter produces the sequence: 1000, 0100, 0010, 0001, …



# Johnson Counter

An n-bit counter of this type generates a counting sequence of length 2n

For example a 4-bit counter produces the sequence: 0000, 1000, 1100, 1110, 1111, 0111, 0011, 0001, 0000, …

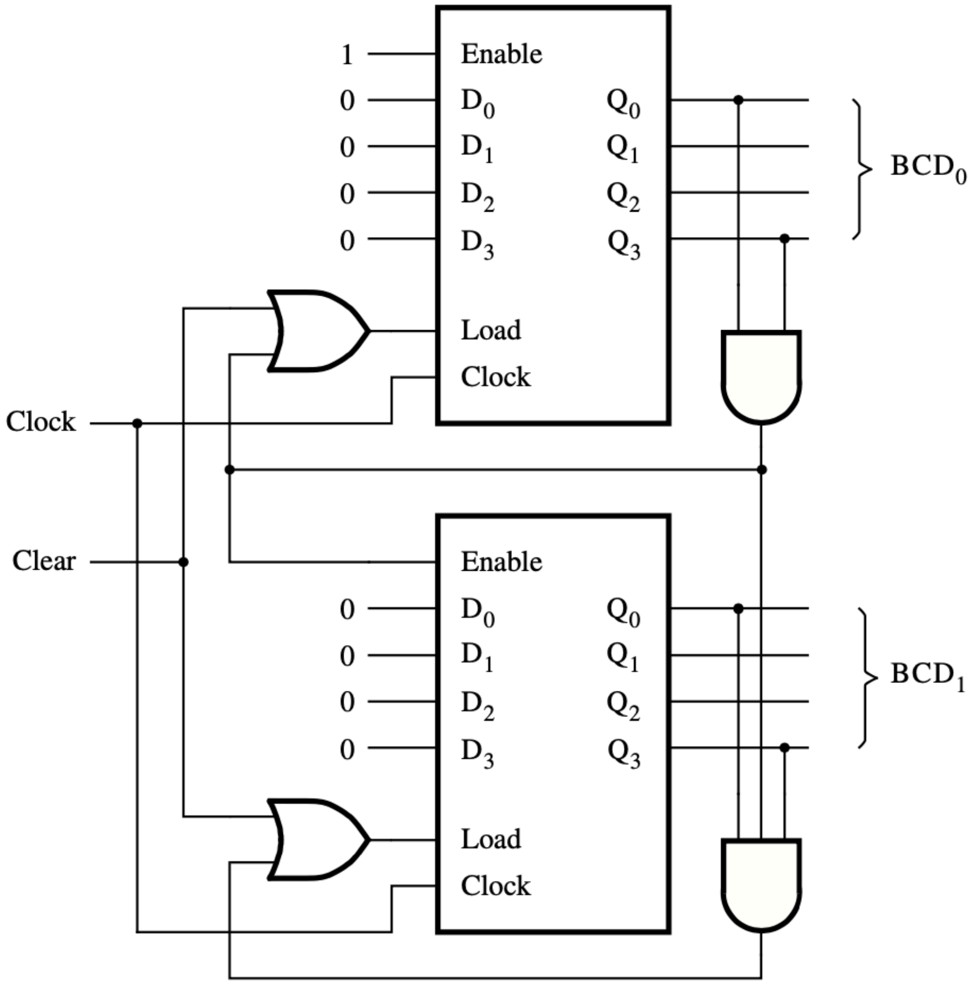


# Next Time

- Finite State Machines
  - Mealy
  - Moore

# Example: Two digit BCD Counter

*It consists of two modulo-10 counters. One for each BCD digit*



# Finite State Machines

CPEN 230 – Introduction to Digital Logic

# Review

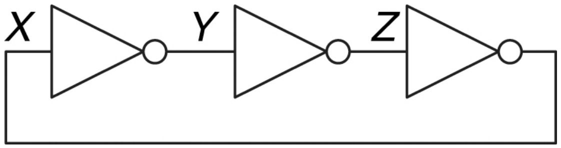
- Synchronous counters
  - Binary Counters (modulo-$2^N$ and modulo-M)
  - Fast counters (Ring counter and Johnson counter)

# Digital Synchronous Systems

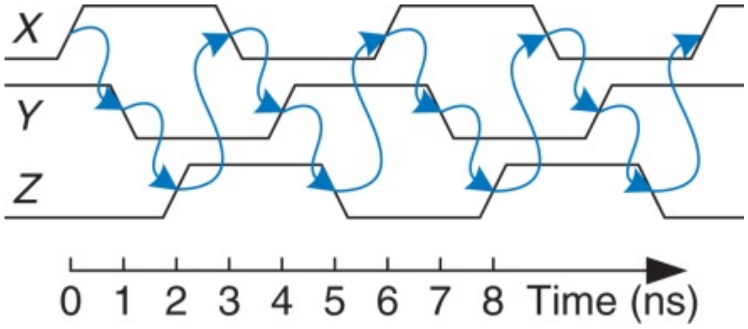
- In general, all circuits that are non combinational are sequential.
- Combinational circuits have no cyclic paths (feedback loops).
- **Cyclic paths can cause undesirable races or unstable behavior.**
- To avoid these problems, designers break the cyclic paths by inserting registers somewhere in the path.
- This way the state of the circuit can change only at the clock edge (is synchronized to the clock).
- **Synchronous sequential circuits:** combinational logic followed by a bank of flip-flops (register).
- Virtually all digital sequential systems are synchronous.

# A problematic sequential circuit (astable behavior)

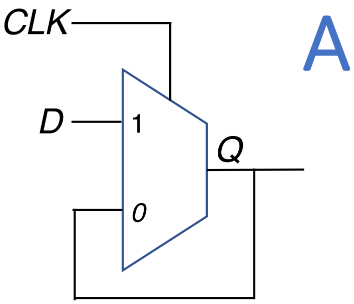
***Ring Oscillator***



Suppose each inverter has a propagation delay of 1 ns ( --> each node oscillates with a period of 6 ns)



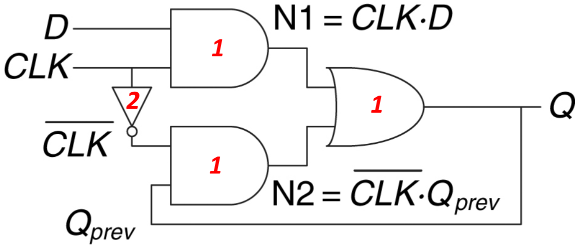
***The circuits has no stable states, so it is said unstable or astable.***

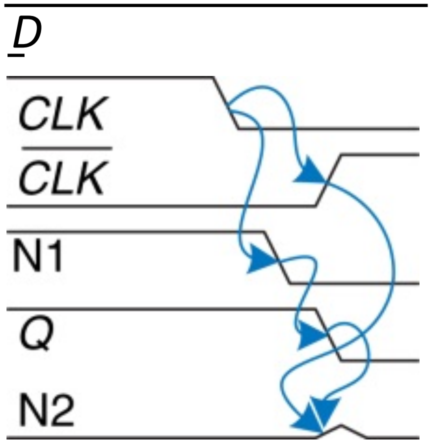


# Another problematic sequential circuit (D-latch with a race condition)

| $CLK$ | $D$ | $Q_{prev}$ | $Q$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

$$Q = CLK\cdot D + \overline{CLK}\cdot Q_{prev}$$





Suppose the delay through the inverter is rather long compared to the delays of the AND an OR gates. Since N1 and Q may both fall before $\overline{\text{CLK}}$ rises , then N2 will never rise, and Q becomes stuck at 0 (despite the latch should remember its old value of Q=1)

# Synchronous Sequential Logic Design

- Breaks cyclic paths by **inserting registers**
- Registers contain **state** of the system
- State changes at clock edge: system **synchronized** to the clock
- **Rules** of synchronous sequential circuit composition:
  - Every circuit element is either a register or a combinational circuit
  - At least one circuit element is a register
  - All registers receive the same clock signal
  - Every cyclic path contains at least one register
- A common synchronous sequential circuit
  - Finite State Machines (FSMs)

# Finite State Machines (FSMs)

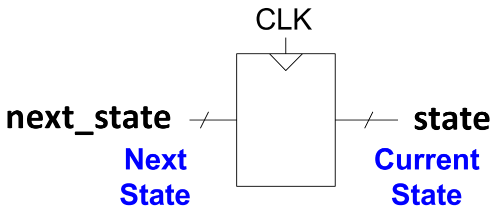
- Three blocks:
  - next state logic (C.L.)

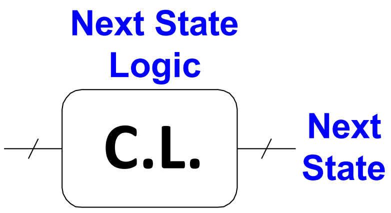
    - Computes the next state
  - state register (S.L.)

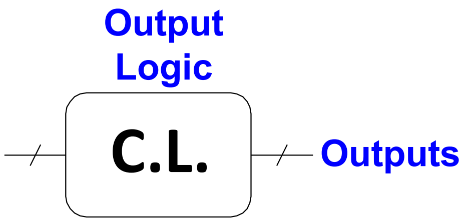
    - Stores current state

    - Loads next state at clock edge
  - output logic (C.L.)

    - Computes the outputs





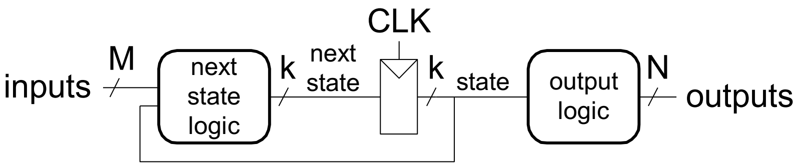


*A circuit with k flip-flops can be in one of a finite number ($2^k$) of unique states*

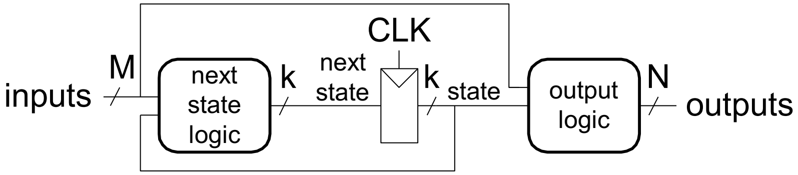
# Finite State Machines (FSMs)

- **Next state** determined by current state and inputs
- Two types of finite state machines differ in **output logic**:
  - **Moore FSM:** outputs depend only on current state
  - **Mealy FSM:** outputs depend on current state *and* inputs

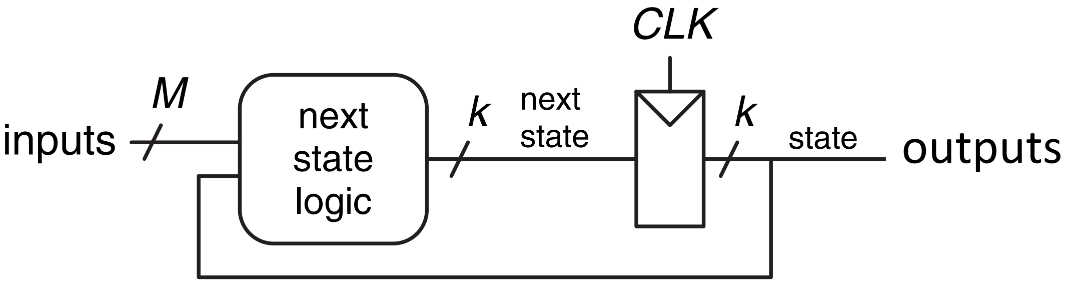
Moore FSM



Mealy FSM



# Counters and FSMs are closely related !

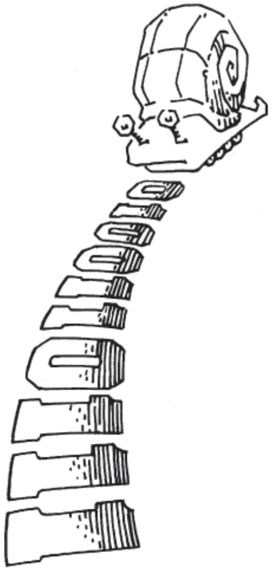


- **Perspective #1:** we can think of a counter as an FSM without the output logic block ( and often with the next state logic having no inputs).

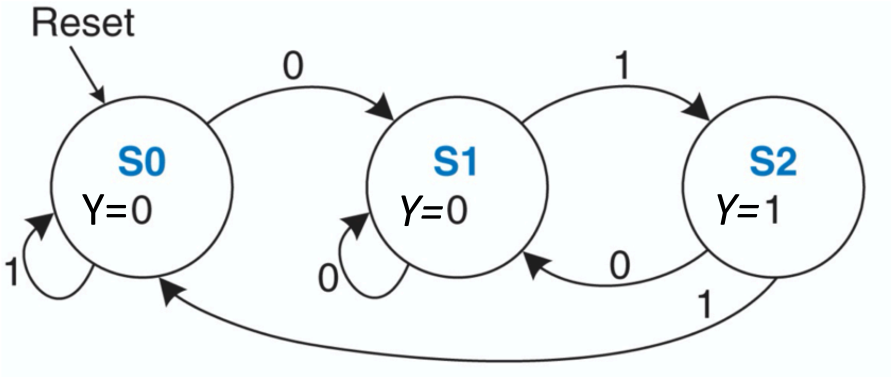
- **Perspective #2:** counters are the “core” of FSMs. They provide the next logic block and the state register block. All it remains for building an FSM out of a counter is adding the output logic block.

# Moore vs. Mealy FSM

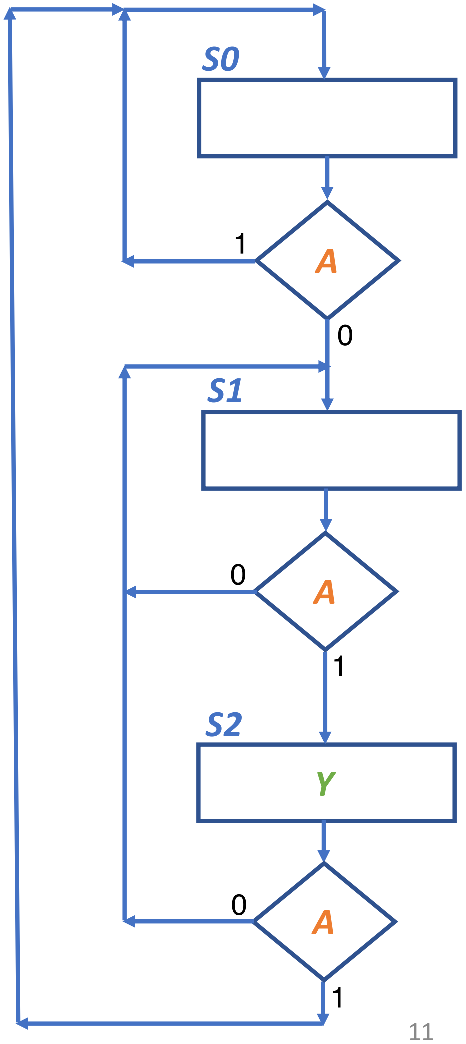
- Maggie T. Hacker has a snail that crawls down a paper tape with 1’s and 0’s on it. The snail smiles whenever the last two digits it has crawled over are 01. Design Moore and Mealy FSMs of the snail’s brain.



# Design Example (Moore FSM)

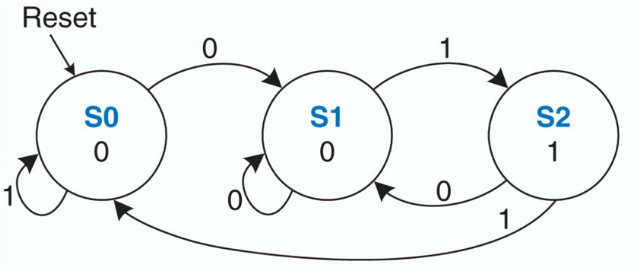


*State Transition Diagram*

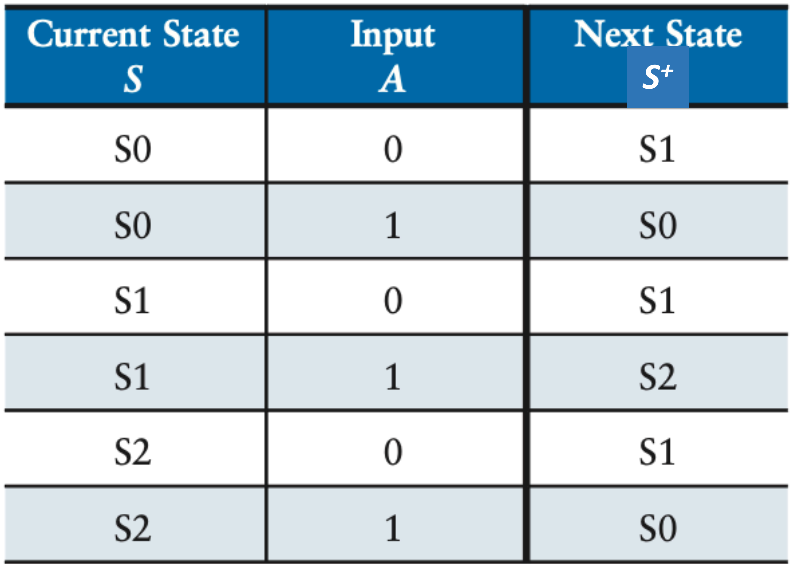


*ASM Diagram*

# Moore FSM Tables

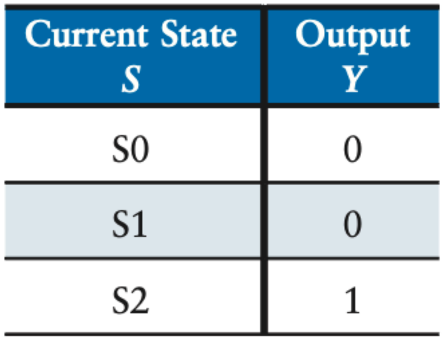


**Moore state transition table**



| Current State<br>$S$ | Input<br>$A$ | Next State<br>$S^+$ |
|---|---|---|
| S0 | 0 | S1 |
| S0 | 1 | S0 |
| S1 | 0 | S1 |
| S1 | 1 | S2 |
| S2 | 0 | S1 |
| S2 | 1 | S0 |

**Moore output table**

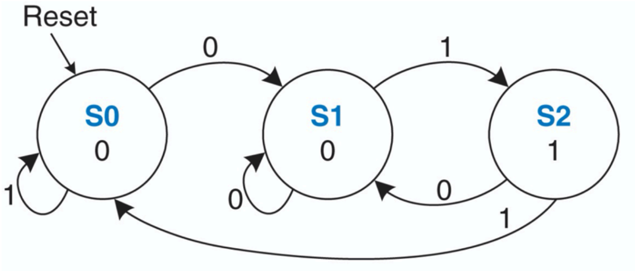


| Current State<br>$S$ | Output<br>$Y$ |
|---|---|
| S0 | 0 |
| S1 | 0 |
| S2 | 1 |

**Binary Encoding of the states**

| State | Encoding |
|---|---|
| S0 | 00 |
| S1 | 01 |
| S2 | 10 |

# Moore FSM Tables with encodings



**Moore state transition table with state encodings**

| Current State | | Input | Next State | |
|---|---|---|---|---|
| $S_1$ | $S_0$ | $A$ | $S_1^+$ | $S_0^+$ |
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 |

**Moore output table with state encodings**

| Current State | | Output |
|---|---|---|
| $S_1$ | $S_0$ | $Y$ |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |

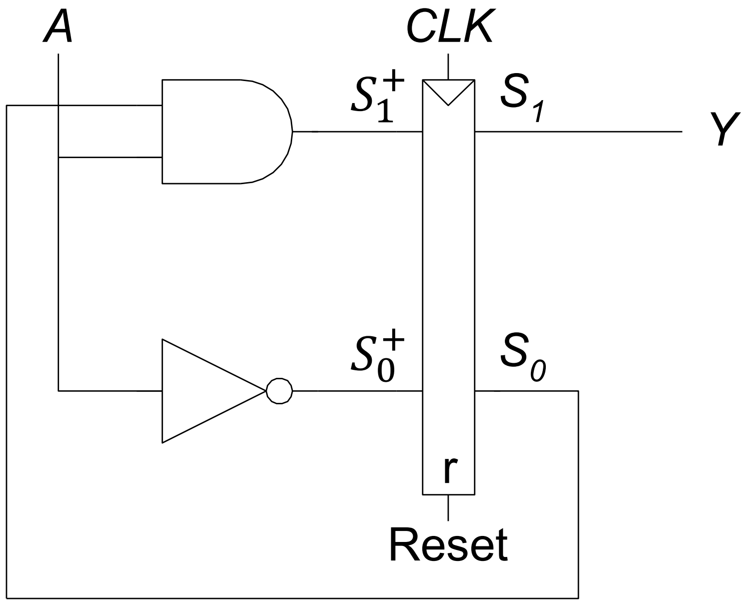
$$S_1^+ = S_0 \cdot A$$

$$S_0^+ = \bar{A}$$

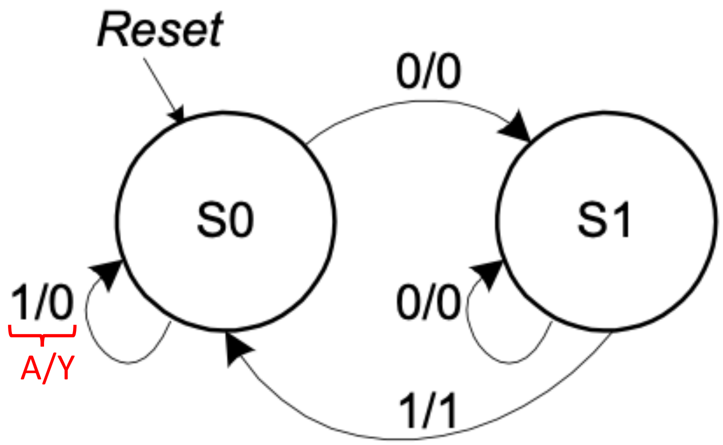
$$Y = S_1$$

# Moore FSM Schematic

$S_1^+ = S_0 \cdot A$ $S_0^+ = \bar{A}$ $Y = S_1$

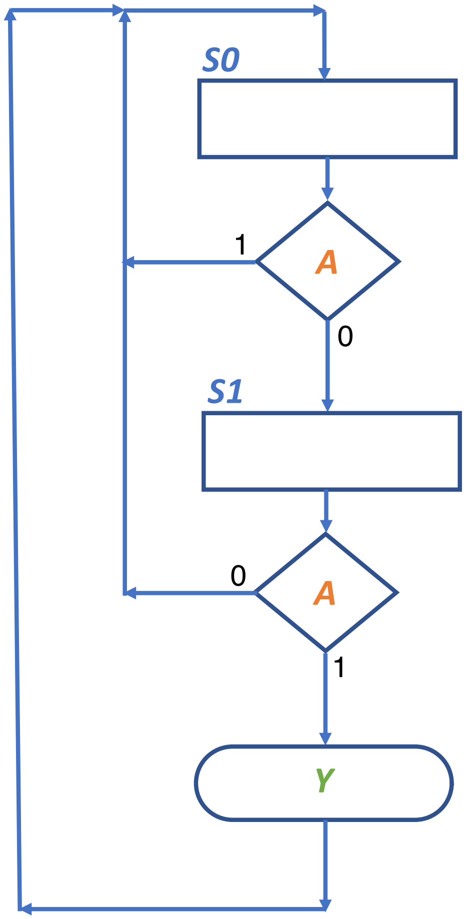


# Design Example (Mealy FSM)

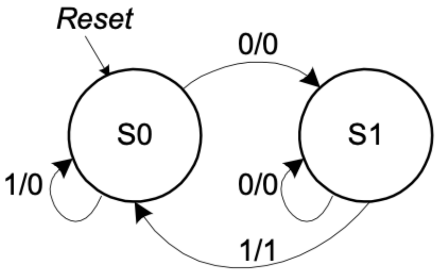


***State Transition Diagram***

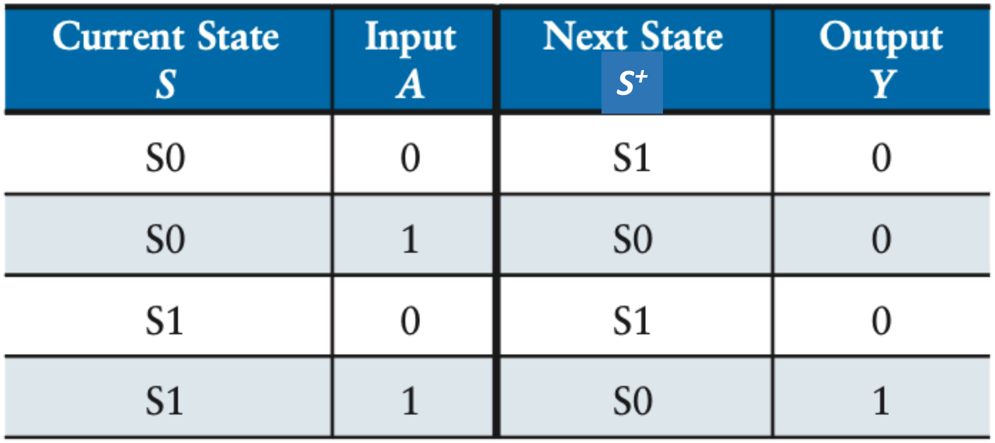
***ASM Diagram***



# Mealy FSM Tables



**Mealy state transition and output table**

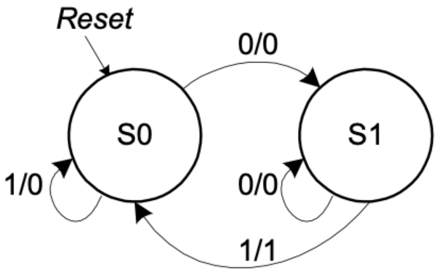


| Current State $S$ | Input $A$ | Next State $S^+$ | Output $Y$ |
|---|---|---|---|
| S0 | 0 | S1 | 0 |
| S0 | 1 | S0 | 0 |
| S1 | 0 | S1 | 0 |
| S1 | 1 | S0 | 1 |

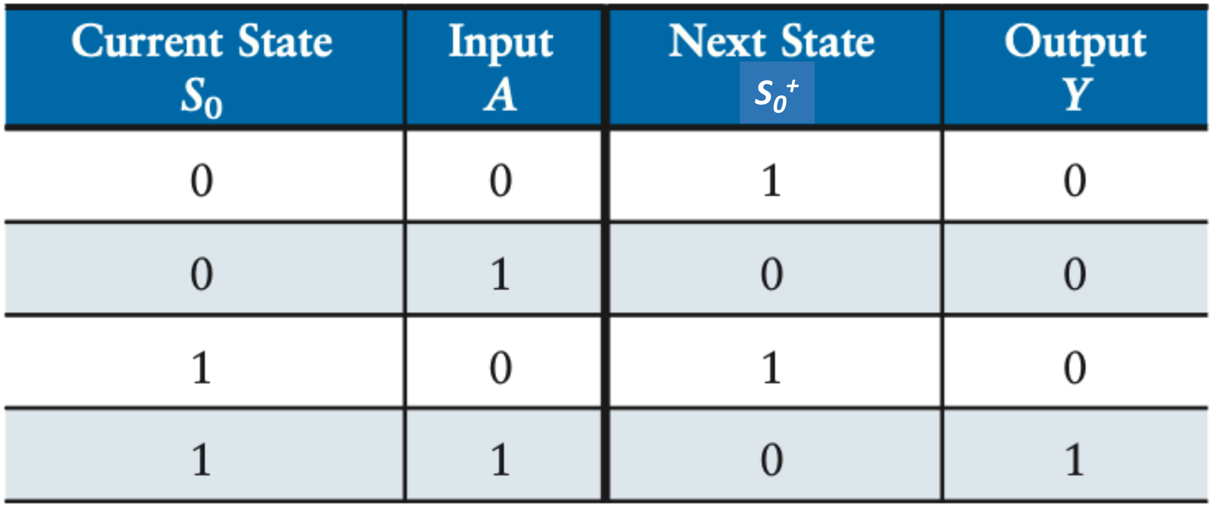
**Binary encoding of the states**

| State | Encoding |
|---|---|
| S0 | 0 |
| S1 | 1 |

# Mealy FSM Tables with encodings



**Mealy state transition and output table with state encodings**



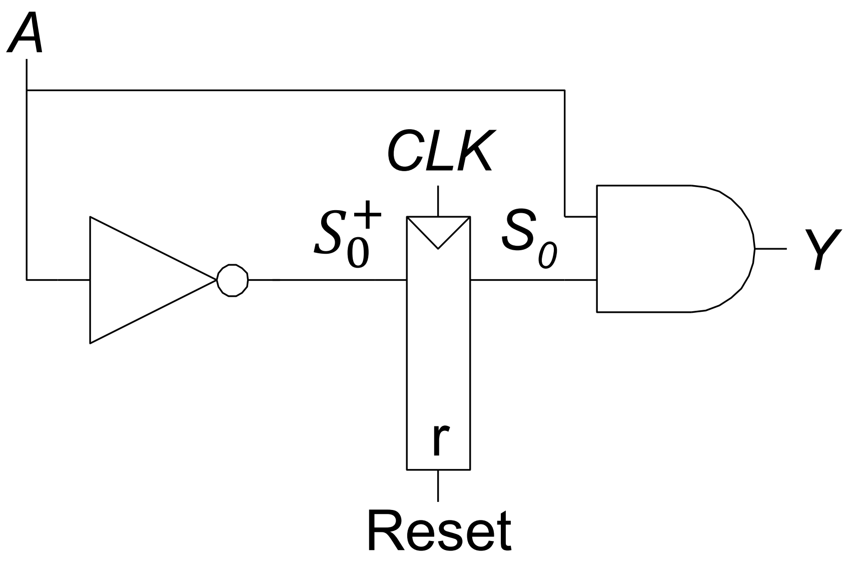
| Current State<br>$S_0$ | Input<br>$A$ | Next State<br>$S_0^+$ | Output<br>$Y$ |
|---|---|---|---|
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

$$S_0^+ = \bar{A}$$

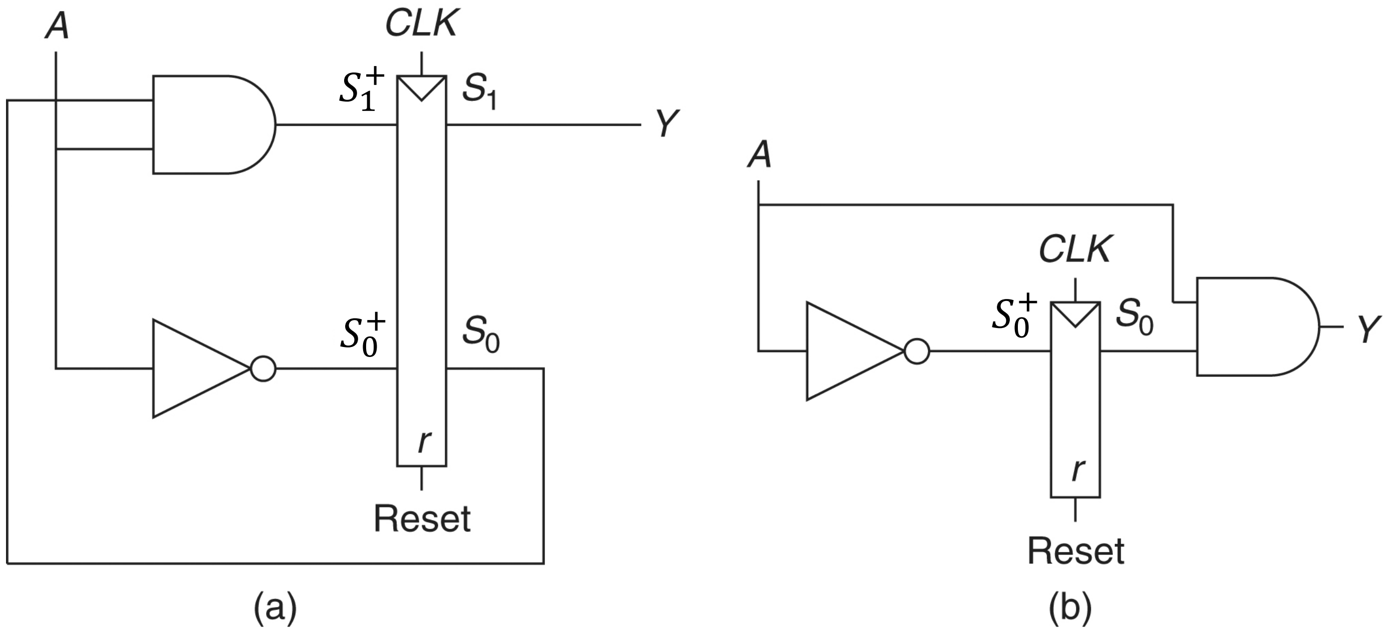
$$Y = S_0 \cdot A$$

# Mealy FSM Schematic

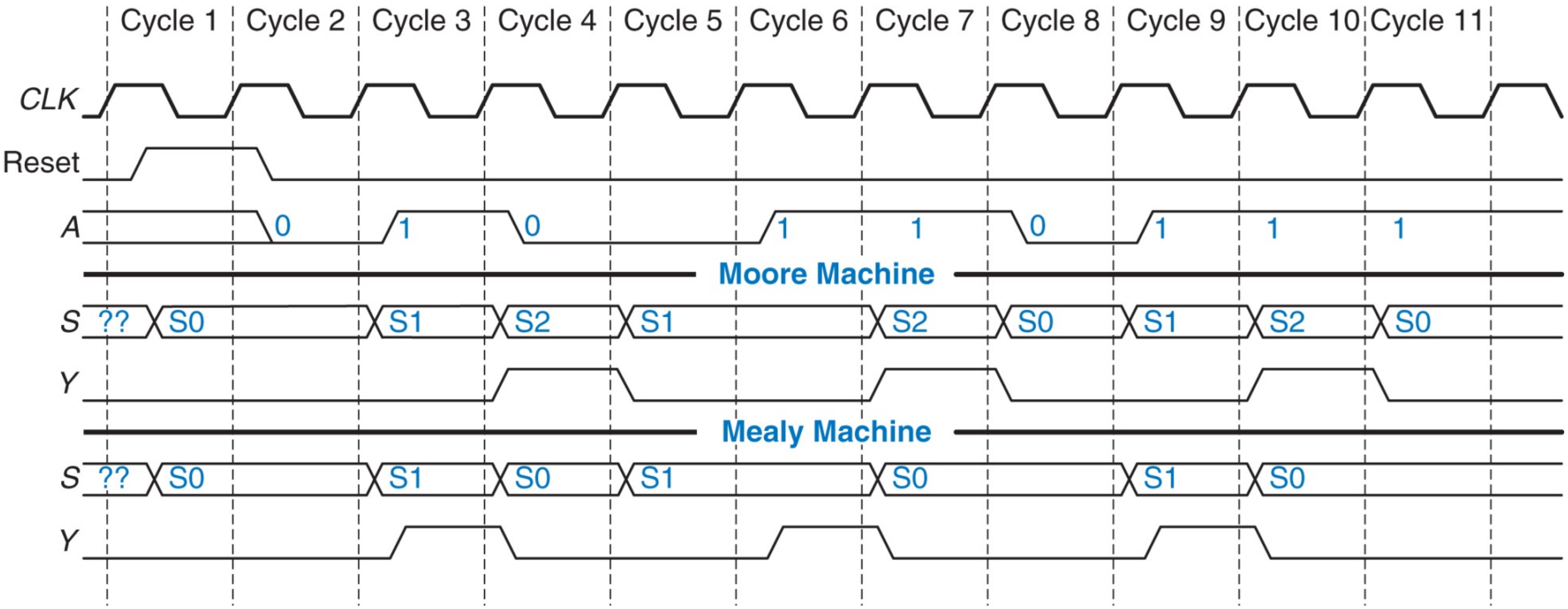
$S_0^+ = \bar{A}$ $Y = S_0 \cdot A$



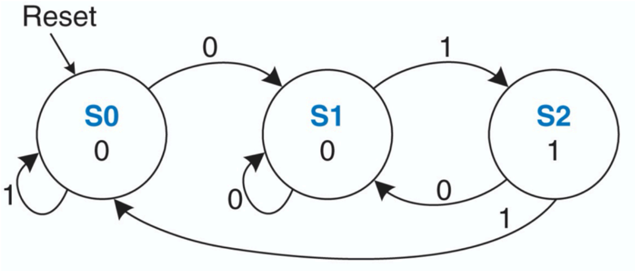
FSM schematics for (a) Moore and (b) Mealy machines



Timing diagrams for Moore and Mealy machines



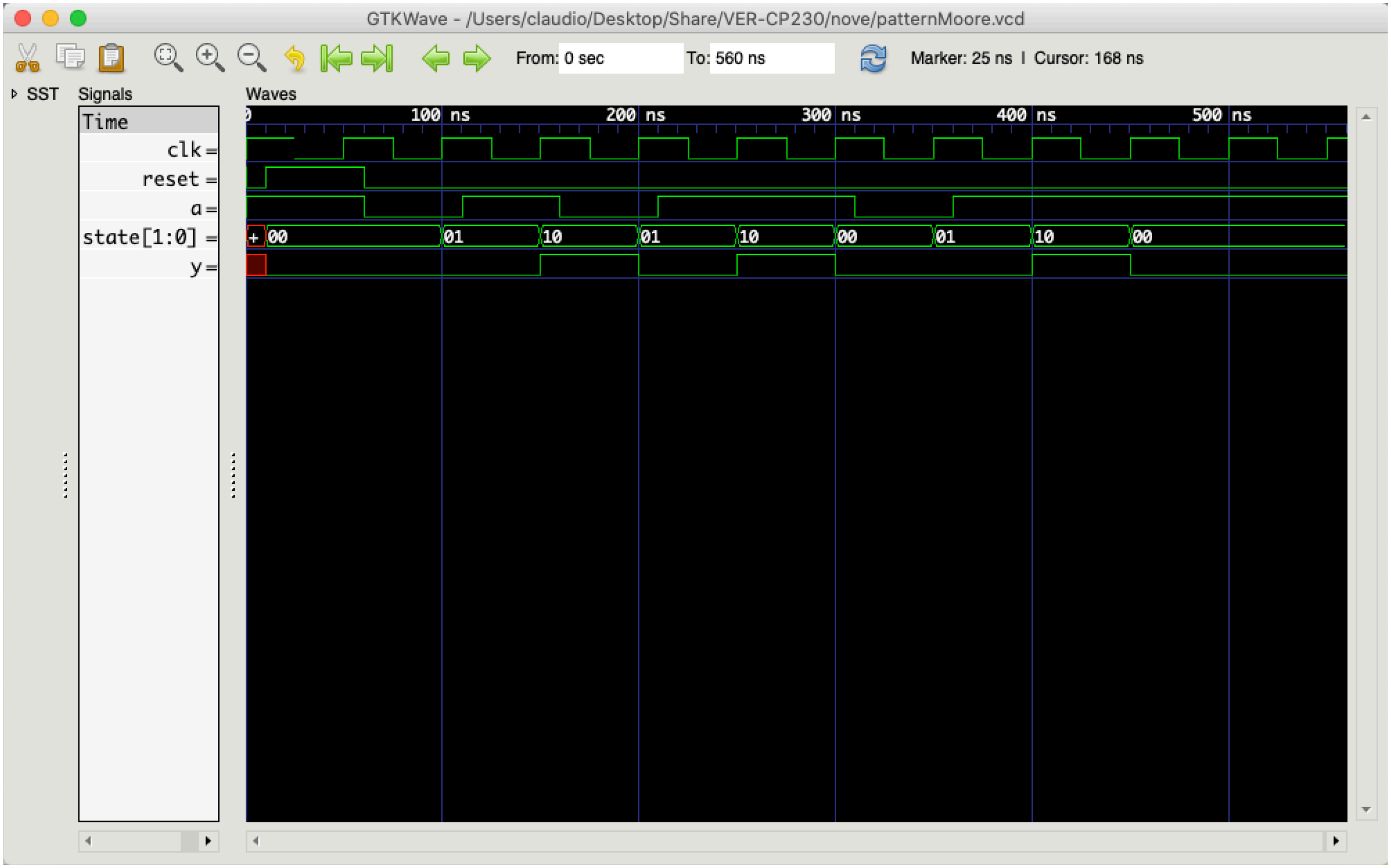
# FSMs in Verilog
# (Pattern Detector: Moore’s FSM)



```
// file: patternMoore.v
// Moore FSM to detect pattern

module patternMoore
(
  input clk,
  input reset,
  input a,
  output reg y
);

  reg [1:0] state, nextstate;

  // declare states
  parameter S0 = 2’b00,
            S1 = 2’b01,
            S2 = 2’b10;

  // state register
  always @(posedge clk, posedge reset) begin
    if (reset) state <= S0;
    else state <= nextstate;
  end
```

```
// next state logic
always @(state, a) begin
  case (state)
    S0: if (a) nextstate = S0;
        else nextstate = S1;
    S1: if (a) nextstate = S2;
        else nextstate = S1;
    S2: if (a) nextstate = S0;
        else nextstate = S1;
    default: nextstate = S0;
  endcase
end

// output logic (output depends only on the state)
always @(state) begin
  if (state == S2)
    y = 1’b1;
  else
    y = 1’b0;
end

endmodule
```

# Pattern Detector: Moore’s FSM



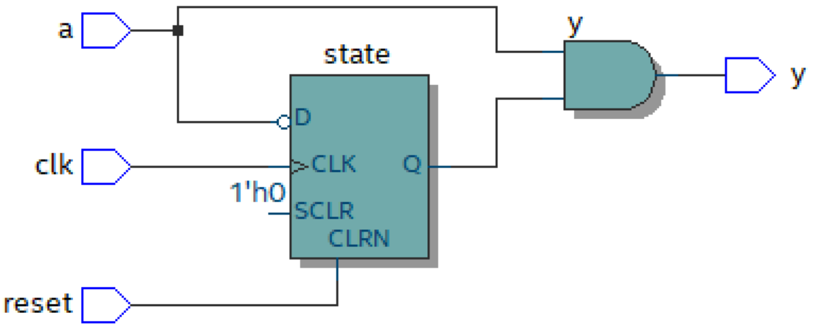
GTKWave - /Users/claudio/Desktop/Share/VER-CP230/nove/patternMoore.vcd

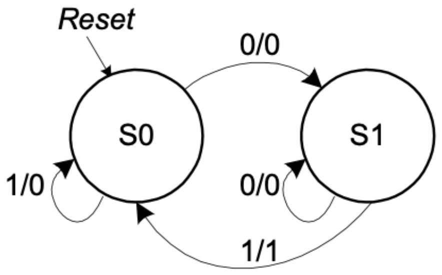
From: 0 sec To: 560 ns

Marker: 25 ns | Cursor: 168 ns

# FSMs in Verilog (Pattern Detector: Mealy's FSM)

```
// file: patternMealy.v
// Mealy FSM to detect pattern

module patternMealy
(
  input clk,
  input reset,
  input a,
  output y
);

  reg state, nextstate;

  // define state
  parameter S0 = 0, S1 = 1;

  // state register
  always @(posedge clk, posedge reset) begin
    if (reset) state <= S0;
    else state <= nextstate;
  end

  // next state logic
  always @(state, a) begin
    case (state)
      S0: if (a) nextstate = S0;
          else nextstate = S1;
      S1: if (a) nextstate = S0;
          else nextstate = S1;
      default: nextstate = S0;
    endcase
  end

  // output logic
  assign y = (a & state == S1);

endmodule
```





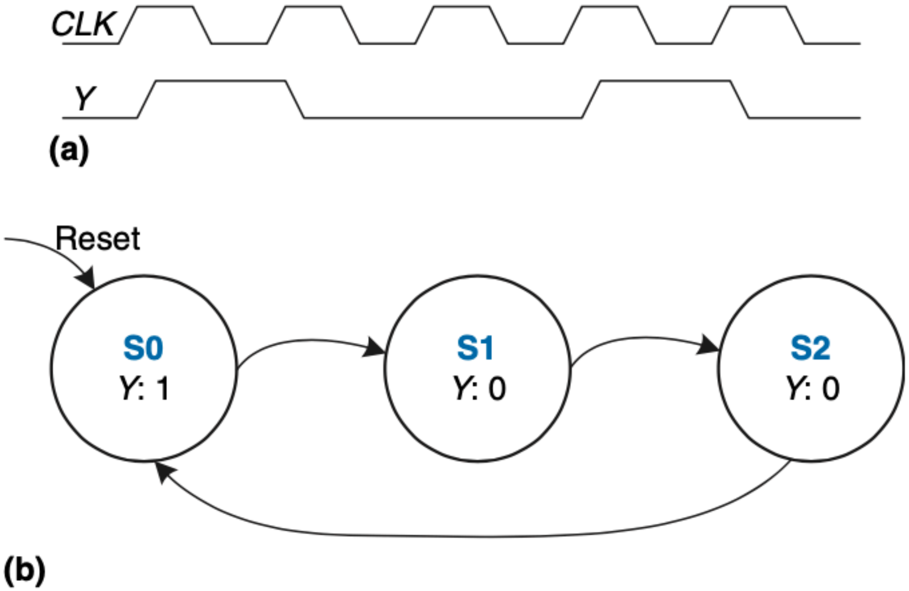
# FSM State Encodings

- **Binary** encoding:
  - i.e., for four states, 00, 01, 10, 11
- **One-hot** encoding
  - One state bit per state
  - Only one state bit HIGH at once
  - i.e., for four states, 0001, 0010, 0100, 1000
  - Requires more flip-flops
  - Often next state and output logic is simpler

# Example: Divide-by-N counter

- Design a divide-by-3 counter using binary and one-hot state encoding

The output Y is HIGH for one clock cycle out of every 3



**Divide-by-3 counter (a) waveform and (b) state transition diagram**

# Divide-by-3 counter

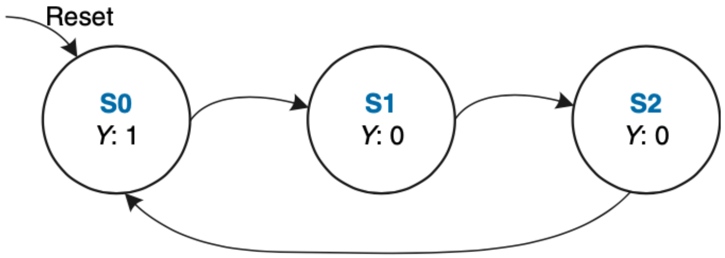
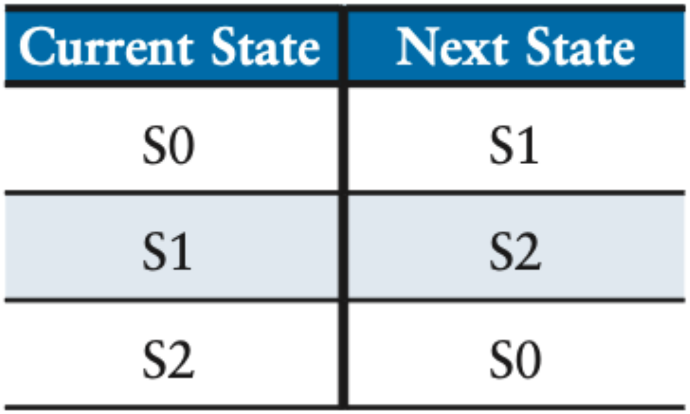
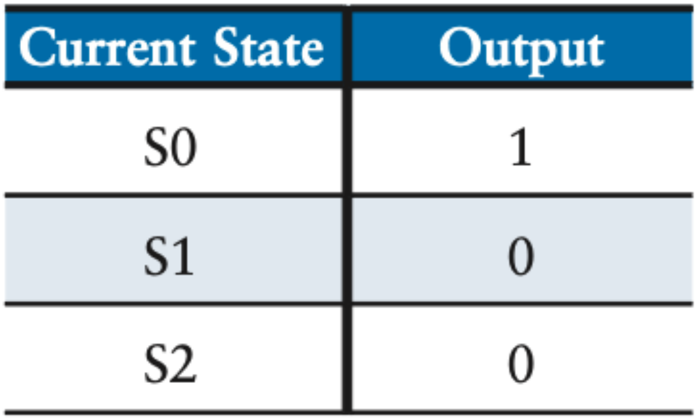


**Table 3.6** **Divide-by-3 counter state transition table**



| Current State | Next State |
|---|---|
| S0 | S1 |
| S1 | S2 |
| S2 | S0 |

**Table 3.7** **Divide-by-3 counter output table**



| Current State | Output |
|---|---|
| S0 | 1 |
| S1 | 0 |
| S2 | 0 |

# Divide-by-3 counter

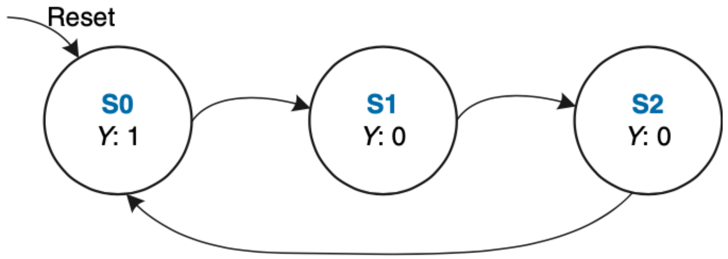
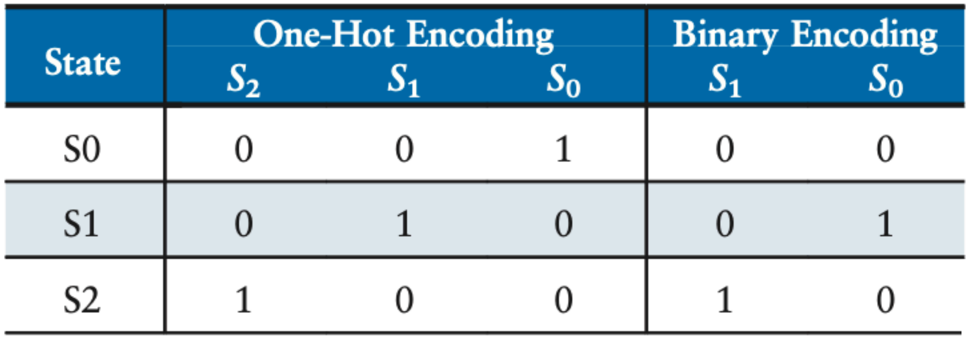
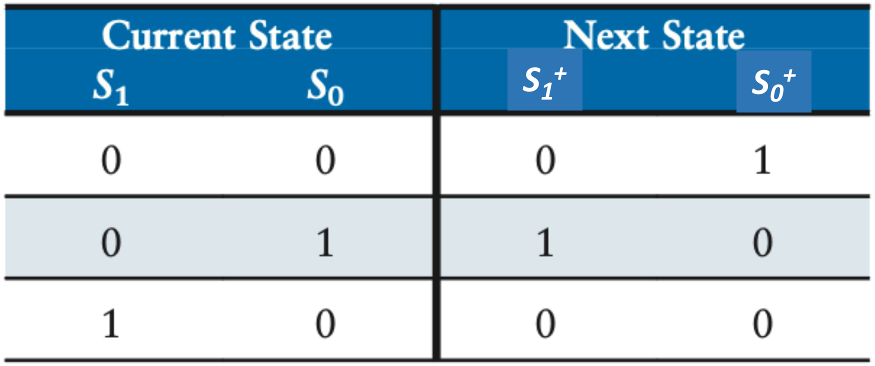


**Table 3.8** One-hot and binary encodings for divide-by-3 counter

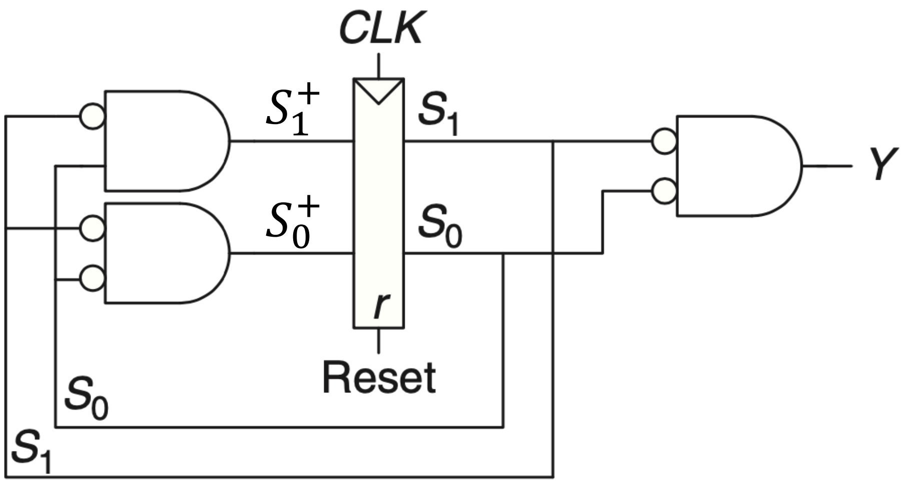


| State | One-Hot Encoding | | | Binary Encoding | |
|---|---|---|---|---|---|
| | $S_2$ | $S_1$ | $S_0$ | $S_1$ | $S_0$ |
| S0 | 0 | 0 | 1 | 0 | 0 |
| S1 | 0 | 1 | 0 | 0 | 1 |
| S2 | 1 | 0 | 0 | 1 | 0 |

**Table 3.9** State transition table with binary encoding



| Current State | | Next State | |
|---|---|---|---|
| $S_1$ | $S_0$ | $S_1^+$ | $S_0^+$ |
| 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 |



# Divide-by-3 counter

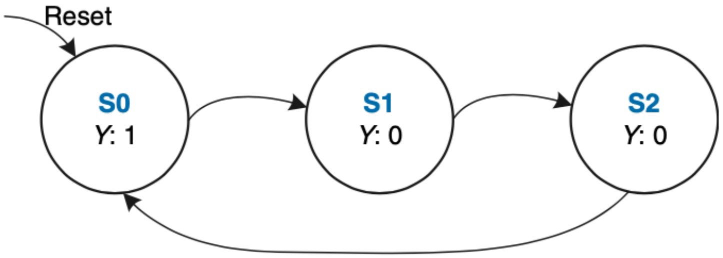
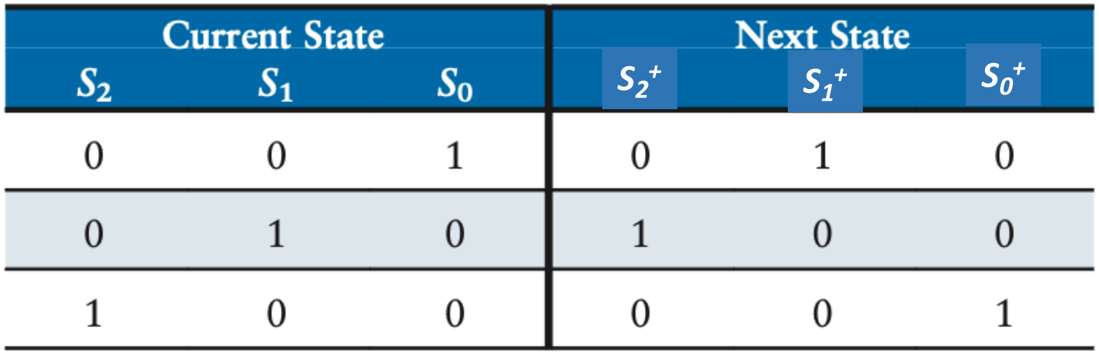
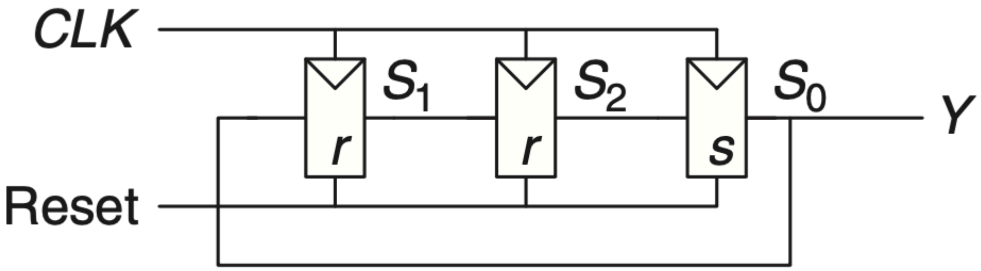


**Table 3.10** **State transition table with one-hot encoding**



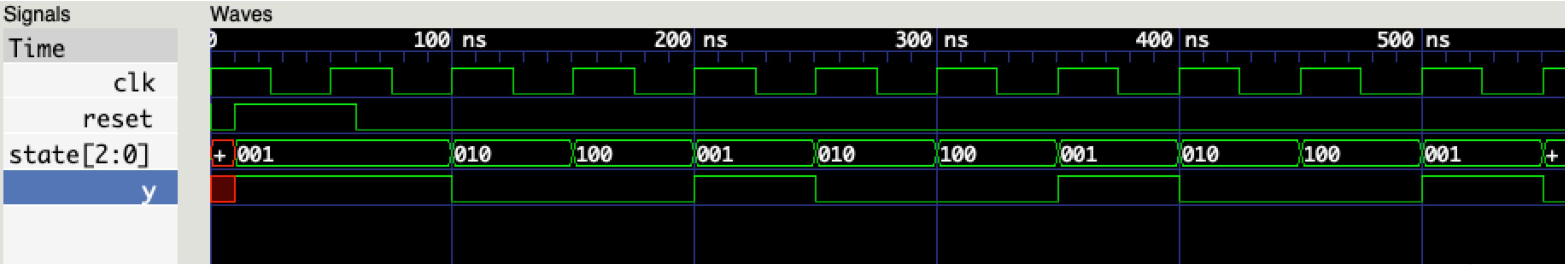
| Current State | | | Next State | | |
|---|---|---|---|---|---|
| $S_2$ | $S_1$ | $S_0$ | $S_2^+$ | $S_1^+$ | $S_0^+$ |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 |



# Divide-by-3 counter (one-hot encoding) in Verilog

```
// file: divideby3.v
// divide by 3 counter using one-hot encoding

module divideby3
(
  input clk,
  input reset,
  output y
);

  reg [2:0] state, nextstate;

  // define states
  parameter S0 = 3'b001,
            S1 = 3'b010,
            S2 = 3'b100;

  // state register
  always @(posedge clk, posedge reset) begin
    if (reset) state <= S0;
    else state <= nextstate;
  end
```

```
  // next state logic
  always @(state) begin
    case (state)
      S0: nextstate = S1;
      S1: nextstate = S2;
      S2: nextstate = S0;
      default: nextstate = S0;
    endcase
  end

  // output logic
  assign y = (state == S0);

endmodule
```



# Review of FSM Design Procedure

1. Identify inputs and outputs
2. Sketch state transition diagram or ASM diagram
3. Write state transition table
4. Select state encodings
5. For Moore machine:
    a. Rewrite state transition table with state encodings
    b. Write output table
6. For a Mealy machine:
    a. Rewrite combined state transition and output table with state encodings
7. Write Boolean equations for next state and output logic
8. Sketch the circuit schematic

# Next time

- S.L. Timing
    - Setup time
    - Hold Time
    - Clock Skew
    - Metastability

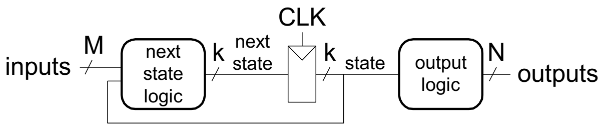
# Timing of Synchronous S.L. circuits

CPEN 230 – Introduction to Digital Logic

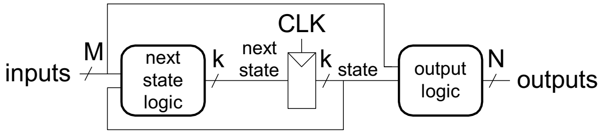
# Review

- Synchronous S.L. circuits
  - Breaks cyclic paths by **inserting registers** (combinational logic followed by register)
- Finite State Machines (FSMs)
  - Three blocks
    - Next state logic (C.L.)
    - State Register (S.L.)
    - Output Logic (C.L.)
- Types of FSMs
  - Moore (outputs depend only on current state)
  - Mealy (outputs depend on current state and current inputs)
- FSM State Encodings
- Coding FSMs in Verilog

Moore FSM



Mealy FSM



# Review: FSM Design

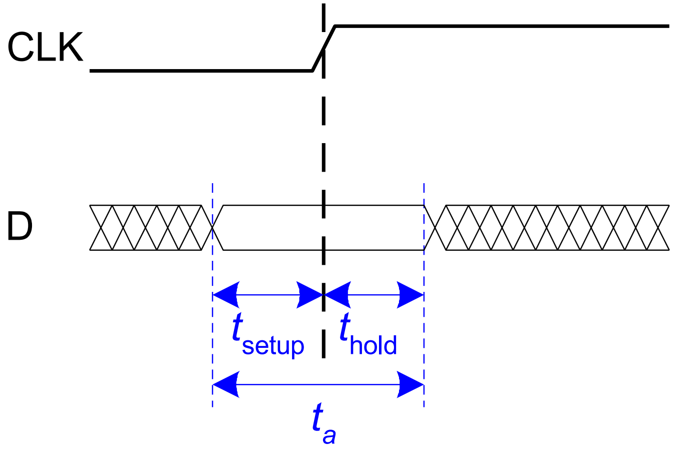
1. Identify inputs and outputs
2. Sketch state transition diagram or ASM diagram
3. Write state transition table
4. Select state encodings
5. For Moore machine:
    a. Rewrite state transition table with state encodings
    b. Write output table
6. For a Mealy machine:
    a. Rewrite combined state transition and output table with state encodings
7. Write Boolean equations for next state and output logic
8. Sketch the circuit schematic

# Timing

- Flip-flop samples *D* at clock edge
- *D* must be stable when sampled
- Similar to a photograph, *D* must be stable around clock edge (both ahead and after the edge = aperture time)
- If not, metastability can occur

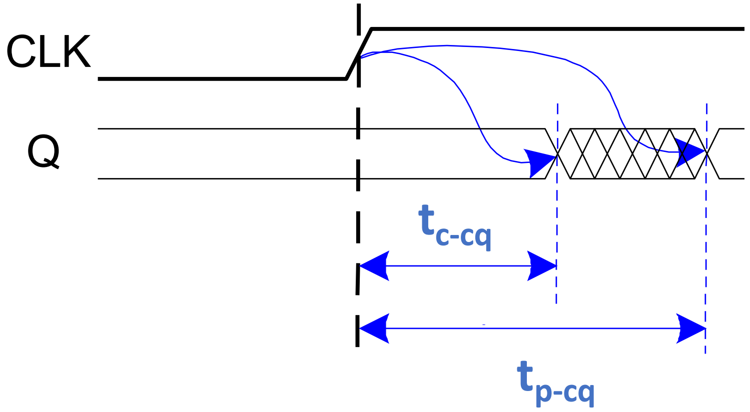
# Input Timing Constrains

- **Setup time:** $t_{setup}$ = time *before* clock edge data must be stable (i.e. not changing)
- **Hold time:** $t_{hold}$ = time *after* clock edge data must be stable
- Aperture time: $t_a$ = time *around* clock edge data must be stable ($t_a = t_{setup} + t_{hold}$)

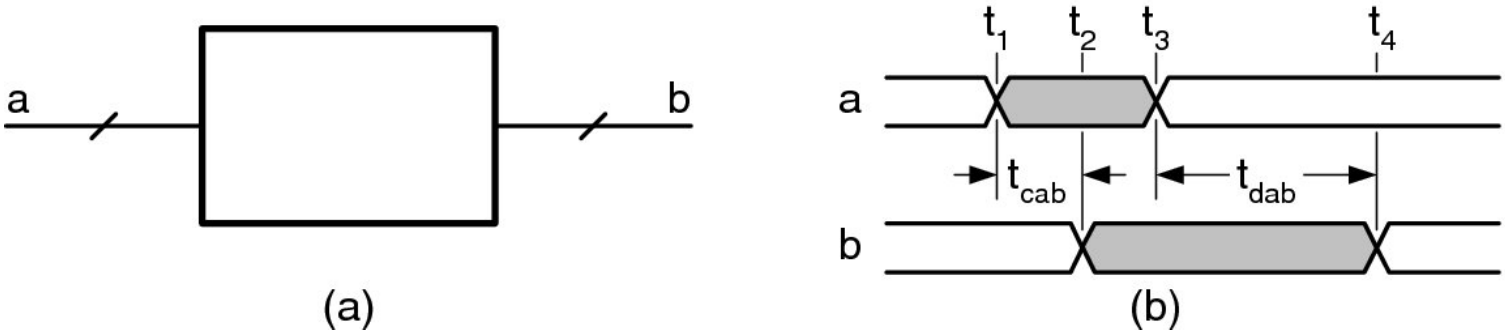


# Output Timing Constrains

- **Propagation delay:** $t_{p\text{-}cq}$ = time after clock edge the output *Q* is guaranteed to be stable (i.e., to stop changing)
- Contamination delay: $t_{c\text{-}cq}$ = time after clock edge *Q* might start being unstable (i.e., start changing)



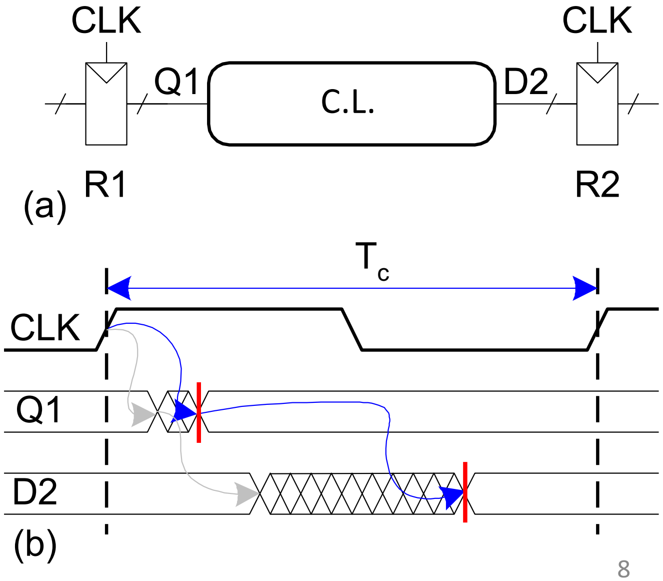
# Propagation delay vs. Contamination delay



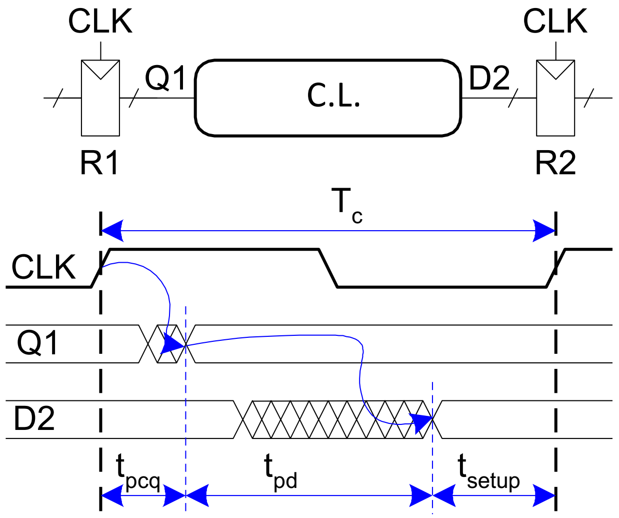
- Propagation delay $t_{d\text{-}ab}$ and contamination delay $t_{c\text{-}ab}$.
- The contamination delay of a logic block is the time from when the first input signal first changes to when the first output signal first changes.
- The propagation delay of a logic block is the time from when the last input signal last changes to when the last output signal last changes

# Dynamic Discipline

- Any generic path in a synchronous sequential circuit, can be mapped in the RTL (register transfer level) structure of figure (a)
- Synchronous sequential circuit inputs must be stable during aperture (setup and hold) time around clock edge. In other words, inputs must be stable
  - at least $t_{setup}$ before the clock edge
  - at least until $t_{hold}$ after the clock edge



# Setup Time Constraint



- The input to register R2 must be stable at least $t_{\text{setup}}$ before clock edge

- It depends on the **maximum delay** it takes to process the signal sampled by R2 (i.e., the maximum delay it takes to go through R1 and the combinational logic)

$$\boxed{T_c \geq t_{p\text{-}cq} + t_{pd} + t_{\text{setup}}}$$

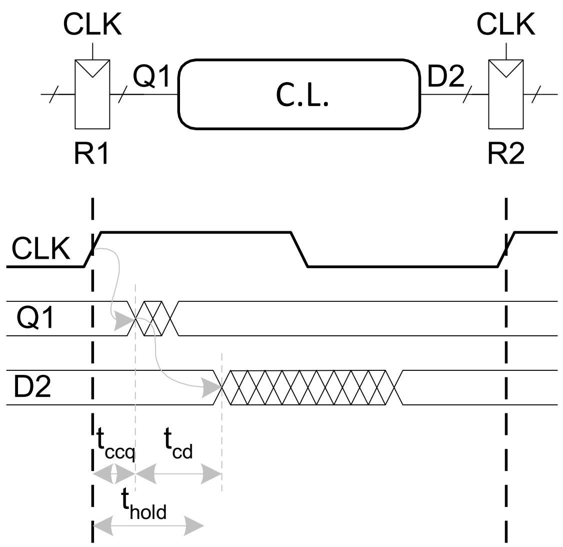
*clock period*

max delay through R1

max delay through C.L.

setup time of R2

# Hold Time Constraint



- The input to register R2 must be stable for at least $t_{\text{hold}}$ after the clock edge
- It depends on the **minimum** delay it takes to go though register R1 and the combinational logic

$$\boxed{t_{\text{hold}} < t_{c\text{-}cq} + t_{c\text{-}d}}$$

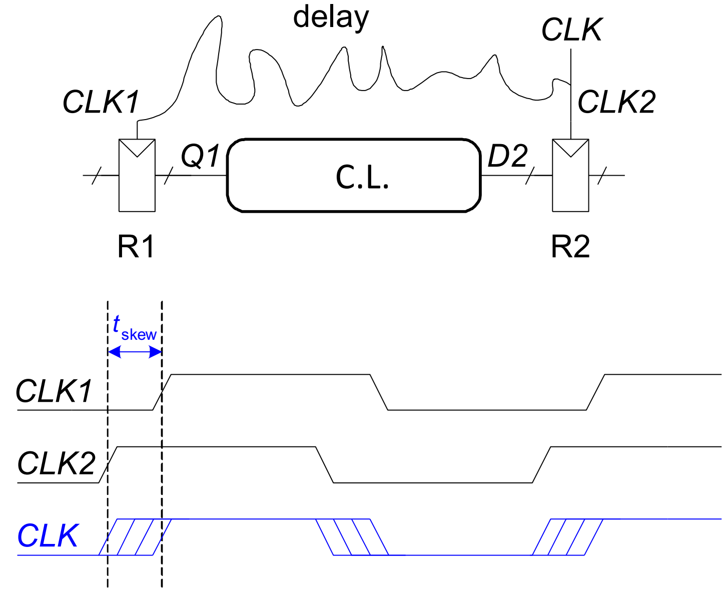
hold time of R2

min delay through R1

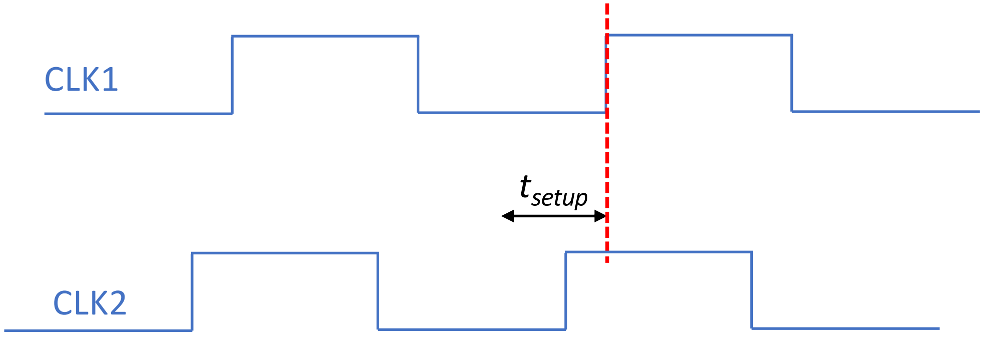
min delay through C.L.

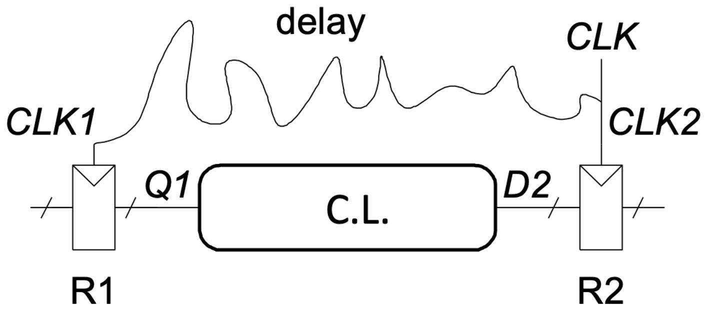
# The effect of clock skew

- The clock doesn’t arrive at all registers at same time
- **Skew:** difference between two clock edges
- Perform **most problematic case analysis** to guarantee dynamic discipline is not violated for any register (many registers in a system!)



# Setup Time constraint with skew

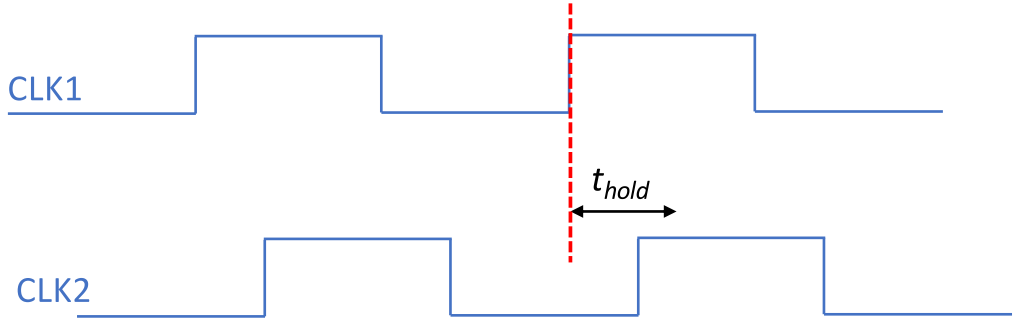


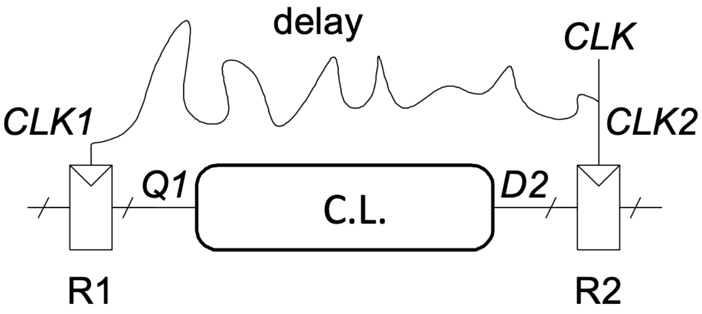


- The most problematic scenario occurs when CLK2 is earlier

$$T_c \geq t_{p\text{-}cq} + t_{pd} + t_{\text{setup}} + t_{skew}$$

# Hold Time constraint with skew



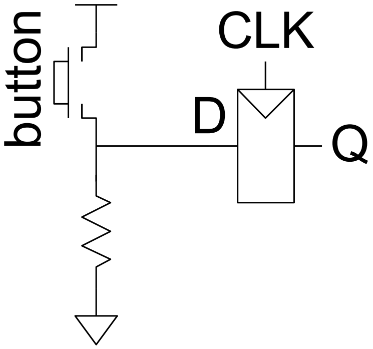


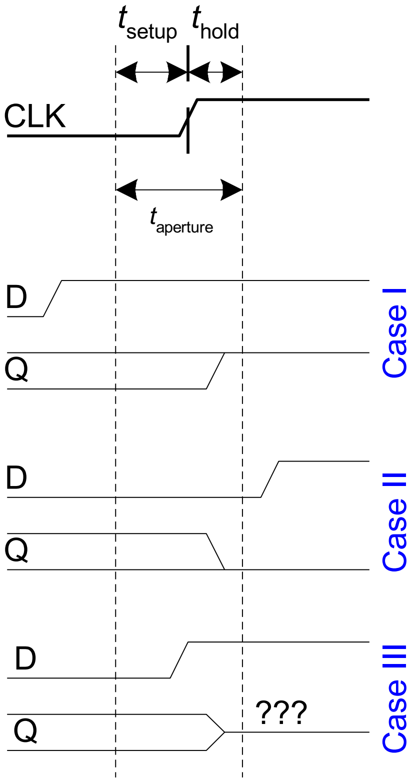
- The most problematic scenario occurs when CLK2 is later

$$t_{hold} \leq t_{c\text{-}cq} + t_{c\text{-}d} - t_{skew}$$

# Violating the Dynamic Discipline

**Asynchronous inputs (for example, user inputs)** might violate the dynamic discipline



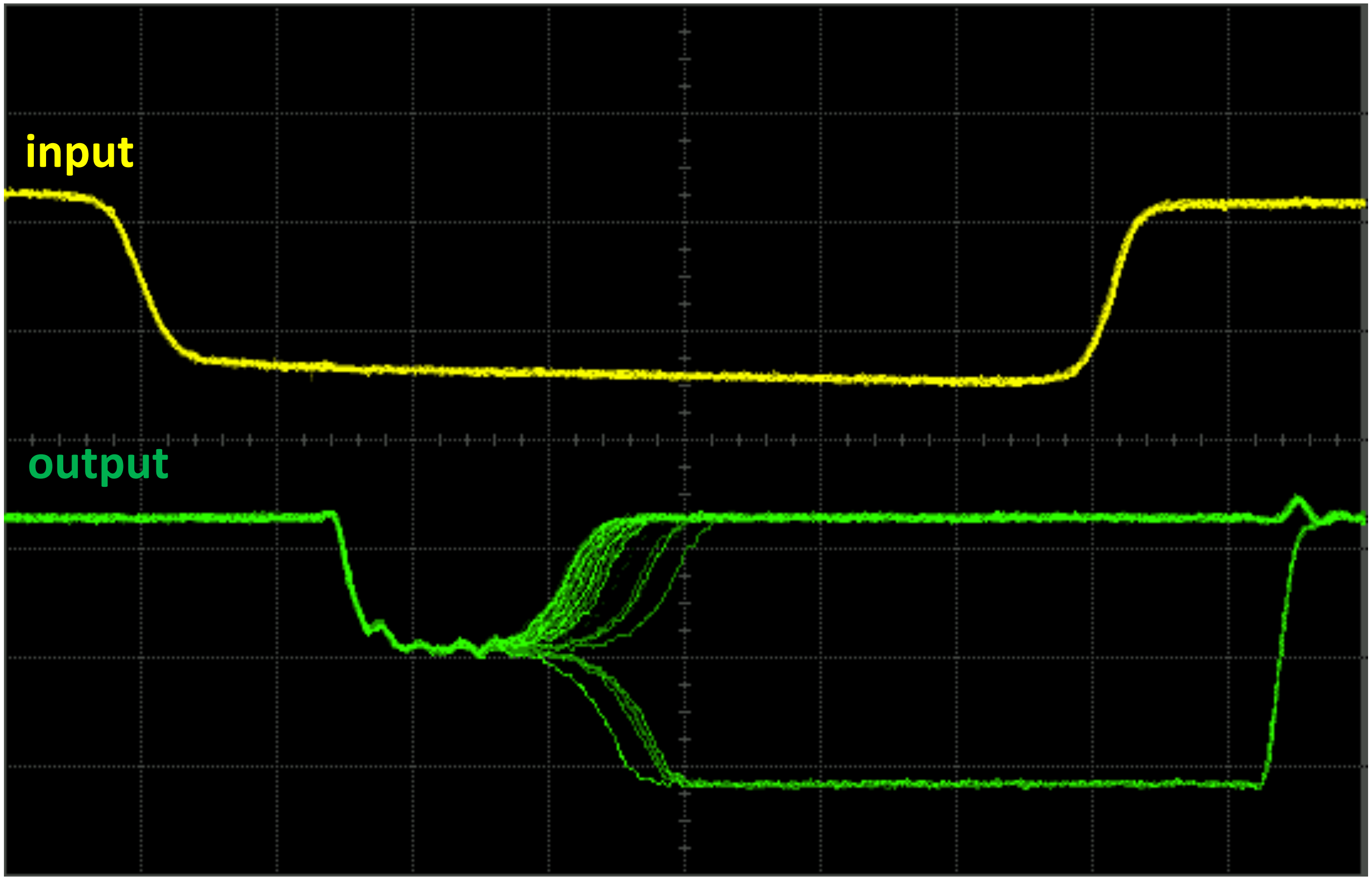


*O.K.*

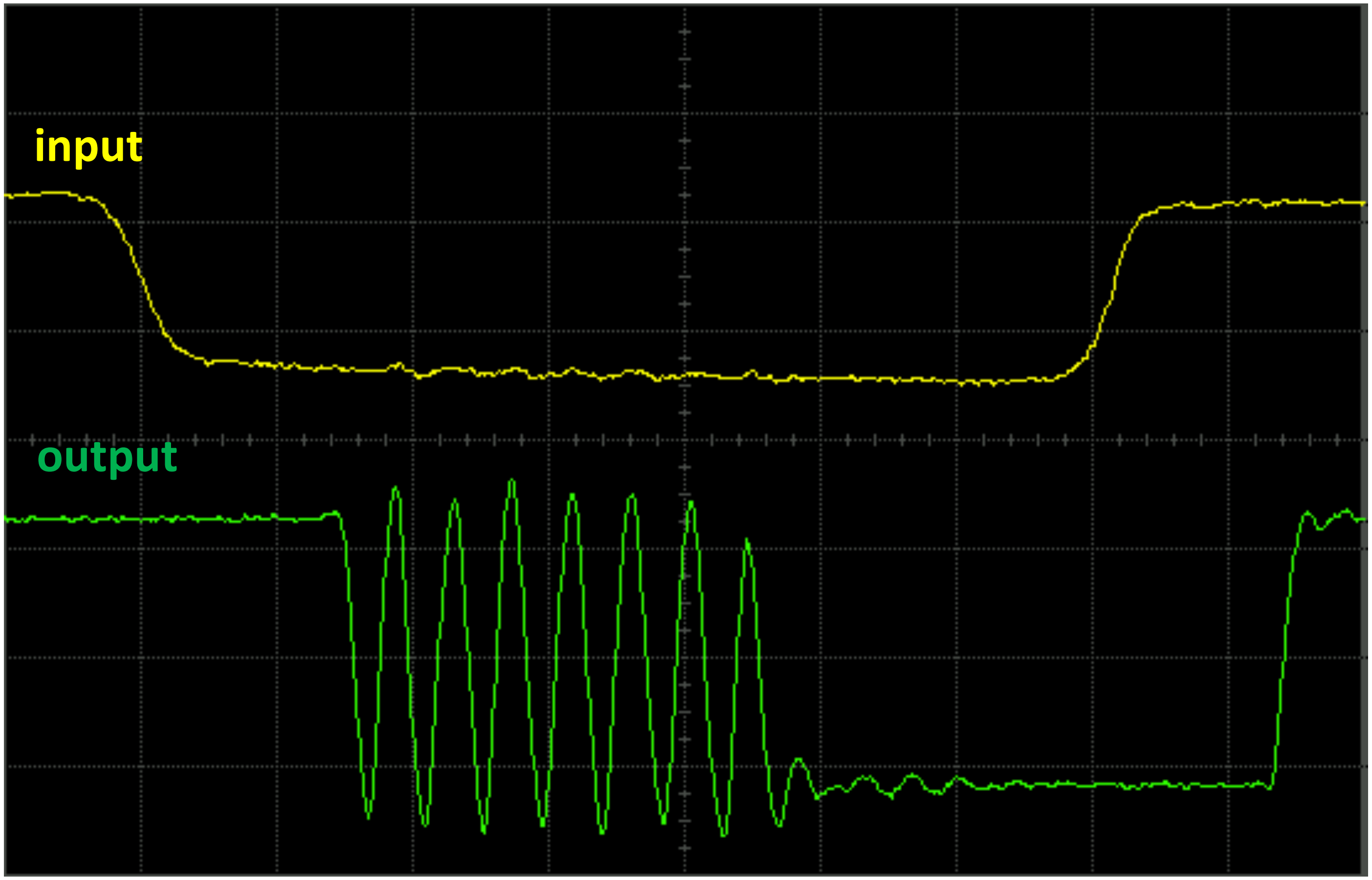
*MISS*

*Metastability*

# Metastability: Example



# Metastability: Example



# One Last FSM Design Example

CPEN 230 – Introduction to Digital Logic

# FSM Design

1. Identify inputs and outputs
2. Sketch state transition diagram or ASM diagram
3. Write state transition table
4. Select state encodings
5. For Moore machine:
   a. Rewrite state transition table with state encodings
   b. Write output table
6. For a Mealy machine:
   a. Rewrite combined state transition and output table with state encodings
7. Write Boolean equations for next state and output logic
8. Sketch the circuit schematic

# Example: Design a Rising-edge Detector

- A rising-edge detector is a circuit that generates a one-clock cycle high level value when the input signal change from 0 to 1.

In class Activity