

EEM 232 Digital System I

Instructor : Assist. Prof. Dr. Emin Germen
egermen@anadolu.edu.tr

Course Book : Logic and Computer Design Fundamentals by Mano & Kime Third Ed/Fourth Ed.. Pearson

Grading

- 1st Midterm Exam : 15%
 - 2nd Midterm Exam : 20%
 - Homework & Quiz(es) : 25%
 - Final Exam : 40%
-
- Homework Policy : Must return in 1 week
(If cheating has been detected the grade will be 0)

Lecture Overview

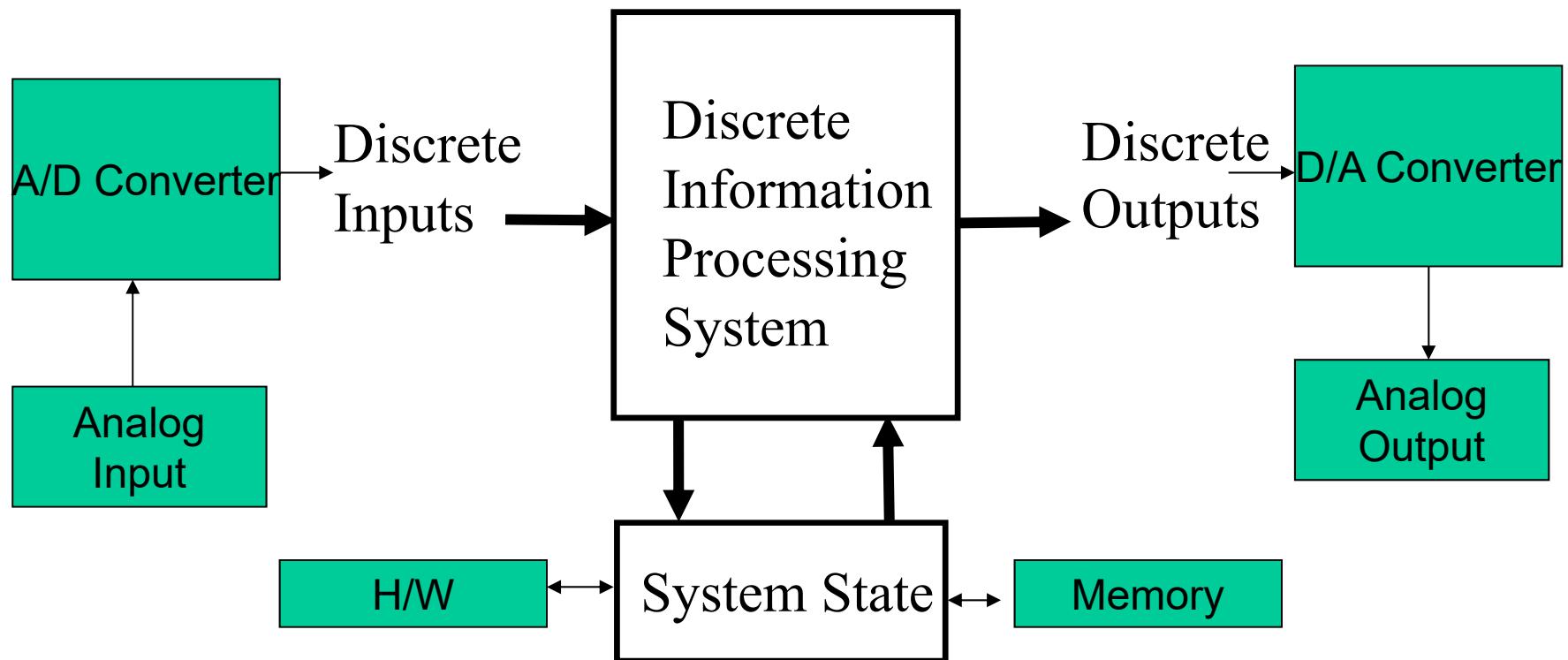
- Week 1 Introduction & Number Systems
- Week 2 Boolean Algebra
- Week 3 Karnaugh Maps
- Week 4 Combinational Logic Design
- Week 5 Combinational Logic Design (Ctd...)
- Week 6 Arithmetic Functions
- Week 7 Sequential Circuits & FlipFlops, Latches
- Week 8 Sequential Circuit Analyses
- Week 9 Sequential Circuits Counters
- Week 10 Sequential Circuit Design
- Week 11 ALU (arithmetic Logic Unit)
- Week 12 RAM

Overview

- Digital Systems and Computer Systems
- Information Representation
- Number Systems [binary, octal and hexadecimal]
- Arithmetic Operations
- Base Conversion
- Decimal Codes [BCD (binary coded decimal), parity]
- Gray Codes
- Alphanumeric Codes

Digital System

Takes a set of discrete information inputs and discrete internal information (system state) and generates a set of discrete information outputs.

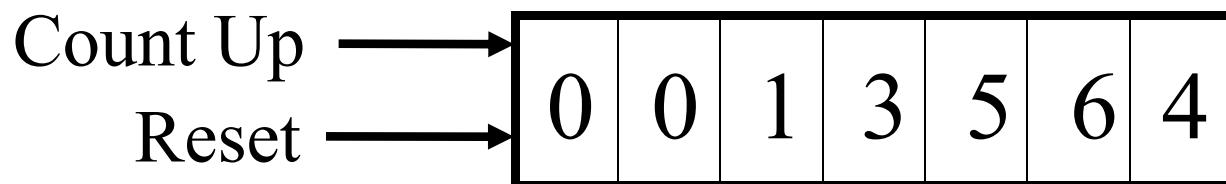


Types of Digital Systems

- No state present
 - Combinational Logic System
 - Output = Function(Input)
- State present
 - State updated at discrete times
 - => Synchronous Sequential System
 - State updated at any time
 - => Asynchronous Sequential System
 - State = Function (State, Input)
 - Output = Function (State) or Function (State, Input)

Digital System Example:

A Digital Counter (e. g., odometer):



Inputs: Count Up, Reset

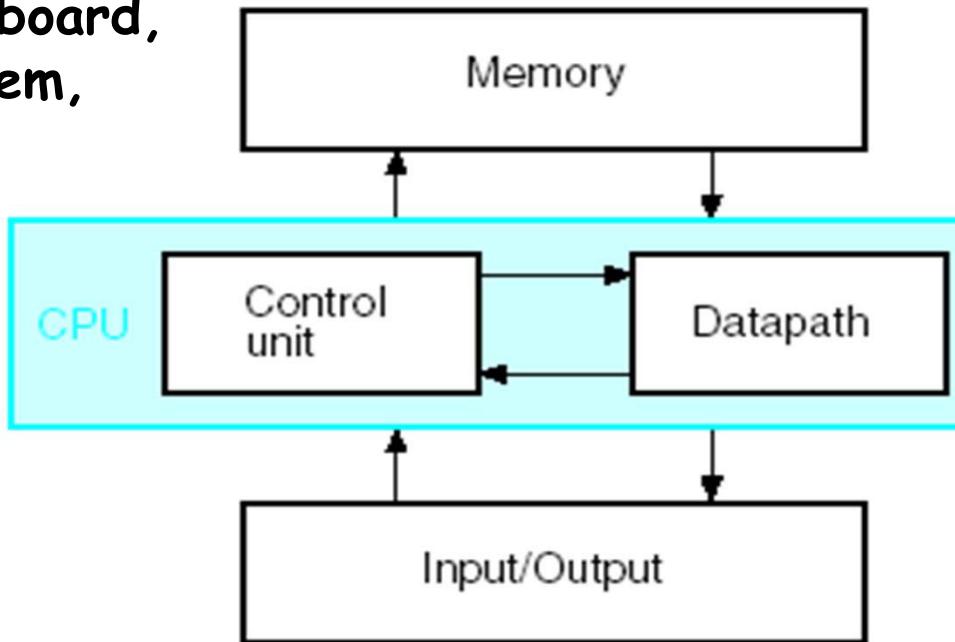
Outputs: Visual Display

State: "Value" of stored digits

Synchronous or Asynchronous?

A Digital Computer Example

Inputs: Keyboard,
mouse, modem,
microphone



Outputs: CRT,
LCD, modem,
speakers

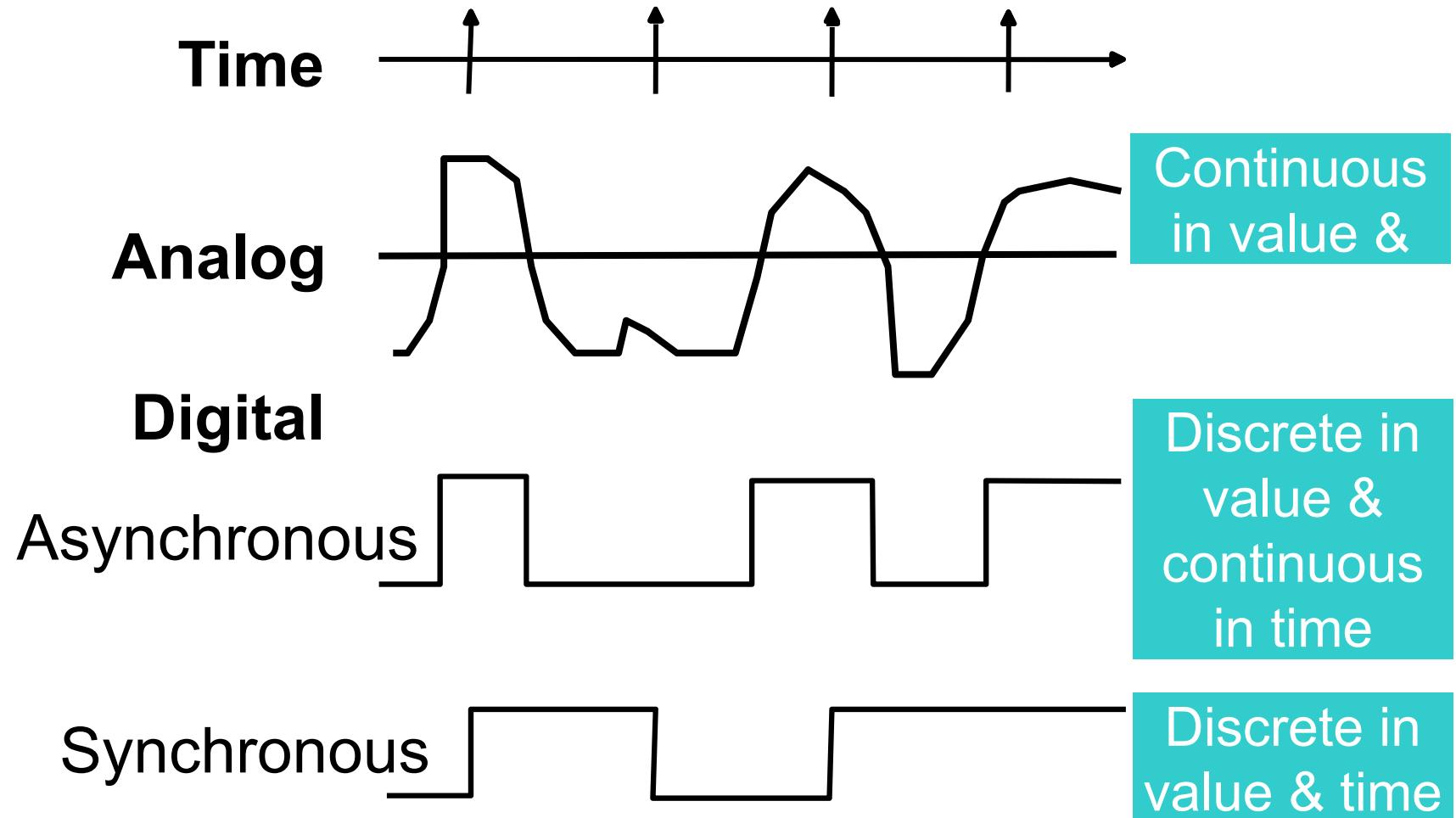
Fig. 1-2 Block Diagram of a Digital Computer

Synchronous or Asynchronous?

Signal

- An information variable represented by physical quantity.
- For digital systems, the variable takes on discrete values.
- Two level, or binary values are the most prevalent values in digital systems.
- Binary values are represented abstractly by:
 - digits 0 and 1
 - words (symbols) False (F) and True (T)
 - words (symbols) Low (L) and High (H)
 - and words On and Off.
- Binary values are represented by values or ranges of values of physical quantities

Signal Examples Over Time



Signal Example – Physical Quantity: Voltage

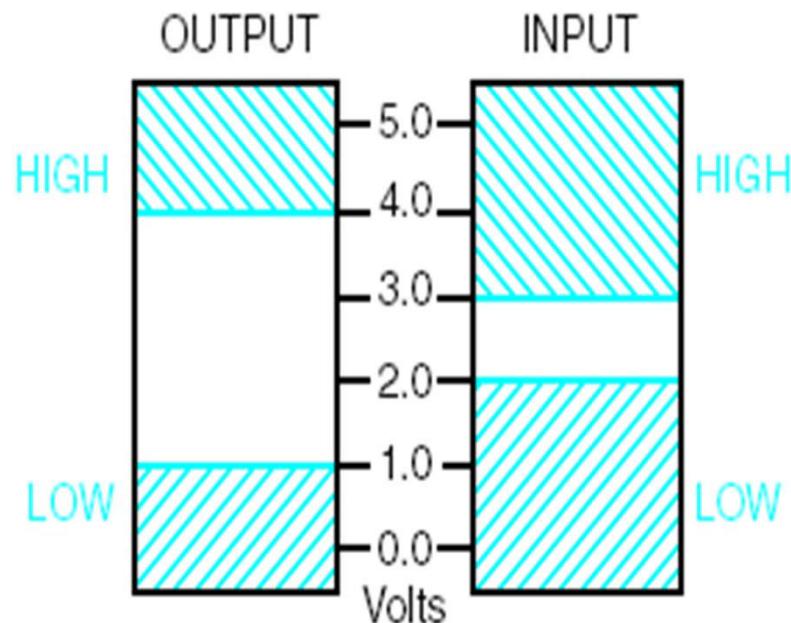


Fig. 1-1 An Example of Voltage Ranges for Binary Signals

Binary Values: Other Physical Quantities

- What are other physical quantities represent 0 and 1?
 - CPU Voltage
 - Disk Magnetic Field Direction
 - CD Surface Pits/Light
 - Dynamic RAM Electrical Charge

Number Systems - Representation

- Positive radix, positional number systems
- A number with *radix r* is represented by a string of digits:

$A_{n-1}A_{n-2} \dots A_1A_0 . A_{-1}A_{-2} \dots A_{-m+1}A_{-m}$
in which $0 \leq A_i < r$ and $.$ is the *radix point*.

- The string of digits represents the power series:

$$(\text{Number})_r = \left(\sum_{i=0}^{i=n-1} A_i \cdot r^i \right) + \left(\sum_{j=-m}^{j=-1} A_j \cdot r^j \right)$$

(Integer Portion) + (Fraction Portion)

Numbers

$$A_{n-1}A_{n-2}\dots A_1A_0.A_{-1}A_{-2}\dots A_{-m+1}A_{-m}$$

- The . is called the radix point
- A_{n-1} : most significant digit (msd)
- A_{-m} : least significant digit (lsd)

$$(724.5)_{10} = 724.5 = 7 \times 10^2 + 2 \times 10^1 + 4 \times 10^0 + 5 \times 10^{-1}$$

$$1620.375 = 1 \times 10^3 + 6 \times 10^2 + 2 \times 10^1 + 3 \times 10^{-1} + 7 \times 10^{-2} + 5 \times 10^{-3}$$

$$(312.4)_5 = 3 \times 5^2 + 1 \times 5^1 + 2 \times 5^0 + 4 \times 5^{-1} = (82.8)_{10}$$

Number Systems - Examples

	General	Decimal	Binary
Radix (Base)	r	10	2
Digits	$0 \Rightarrow r - 1$	$0 \Rightarrow 9$	$0 \Rightarrow 1$
Power of Radix	r^0	1	1
	r^1	10	2
	r^2	100	4
	r^3	1000	8
	r^4	10,000	16
	r^5	100,000	32
	r^{-1}	0.1	0.5
	r^{-2}	0.01	0.25
	r^{-3}	0.001	0.125
	r^{-4}	0.0001	0.0625
	r^{-5}	0.00001	0.03125

Special Powers of 2

- 2^{10} (1024) is Kilo, denoted "K"
- 2^{20} (1,048,576) is Mega, denoted "M"
- 2^{30} (1,073, 741,824)is Giga, denoted "G"

Positive Powers of 2

- Useful for Base Conversion

Exponent	Value
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

Exponent	Value
11	2,048
12	4,096
13	8,192
14	16,384
15	32,768
16	65,536
17	131,072
18	262,144
19	524,288
20	1,048,576
21	2,097,152

Converting Binary to Decimal

- To convert to decimal, use decimal arithmetic to form Σ (digit \times respective power of 2).
- Example: Convert 11010_2 to N_{10} :

■ Answer : 26

■ How? $1*16 + 1*8 + 0*4 + 1*2 + 0*1$

Commonly Occurring Bases

Name	Radix	Digits
Binary	2	0,1
Octal	8	0,1,2,3,4,5,6,7
Decimal	10	0,1,2,3,4,5,6,7,8,9
Hexadecimal	16	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

The six letters (in addition to the 10 integers) in hexadecimal represent:

Numbers in Different Bases

- Good idea to memorize!

Decimal (Base 10)	Binary (Base 2)	Octal (Base 8)	Hexadecimal (Base 16)
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

Conversion Between Bases

- 1) Convert the Integer Part
- 2) Convert the Fraction Part
- 3) Join the two results with a radix point

Conversion Details

- To Convert the Integral Part:

Repeatedly divide the number by the new radix and save the remainders. The digits for the new radix are the remainders in *reverse order* of their computation. If the new radix is > 10 , then convert all remainders > 10 to digits A, B, ...

- To Convert the Fractional Part:

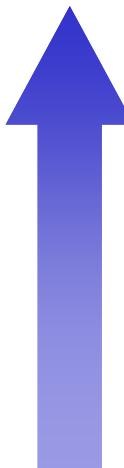
Repeatedly multiply the fraction by the new radix and save the integer digits that result. The digits for the new radix are the integer digits in *order* of their computation. If the new radix is > 10 , then convert all integers > 10 to digits A, B, ...

Example: Convert 162.375_{10} To Base 2

- Convert 162 to Base 2
- Convert 0.375 to Base 2:
- Join the results together with the radix point:

Conversion from binary to decimal

162 / 2 = 81	rem 0
81 / 2 = 40	rem 1
40 / 2 = 20	rem 0
20 / 2 = 10	rem 0
10 / 2 = 5	rem 0
5 / 2 = 2	rem 1
2 / 2 = 1	rem 0
1 / 2 = 0	rem 1



0.375 × 2 = 0.750
0.750 × 2 = 1.500
0.500 × 2 = 1.000



Additional Issue - Fractional Part

- Note that in this conversion, the fractional part became 0 as a result of the repeated multiplications.
- In general, it may take many bits to get this to happen or it may never happen.
- Example: Convert 0.65_{10} to N_2
 - $0.65 = 0.1010011001001 \dots$
 - The fractional part begins repeating every 4 steps yielding repeating 1001 forever!
- Solution: Specify number of bits to right of radix point and round or truncate to this number.

Checking the Conversion

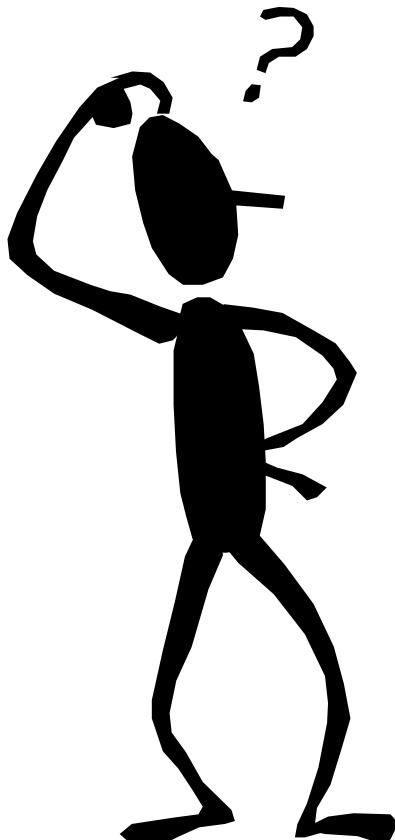
To convert back, sum the digits times their respective powers of r.

- From the prior conversion of 162.375_{10}

$$\begin{aligned}10100010_2 &= 1 \cdot 128 + 0 \cdot 64 + 1 \cdot 32 + 0 \cdot 16 + 0 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 \\&= 128 + 32 + 2 \\&= 162\end{aligned}$$

$$\begin{aligned}0.011_2 &= 1/4 + 1/8 \\&= 0.250 + 0.1250 \\&= 0.375\end{aligned}$$

Why does this work?



- This works for converting from decimal to *any* base
- Why? Think about converting 162.375 from decimal to decimal

$$\begin{aligned}162 / 10 &= 16 \quad \text{rem } 2 \\16 / 10 &= 1 \quad \text{rem } 6 \\1 / 10 &= 0 \quad \text{rem } 1\end{aligned}$$

- Each division strips off the rightmost digit (the remainder). The quotient represents the remaining digits in the number
- Similarly, to convert fractions, each multiplication strips off the leftmost digit (the integer part). The fraction represents the remaining digits

$$\begin{aligned}0.375 \times 10 &= 3.750 \\0.750 \times 10 &= 7.500 \\0.500 \times 10 &= 5.000\end{aligned}$$

Octal and Hexadecimal Numbers

- The octal number system: Base-8
- Eight digits: 0,1,2,3,4,5,6,7

$$(127.4)_8 = 1 \times 8^2 + 2 \times 8^1 + 7 \times 8^0 + 4 \times 8^{-1} = (87.5)_{10}$$

- The hexadecimal number system: Base-16
- Sixteen digits: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

$$(B65F)_{16} = 11 \times 16^3 + 6 \times 16^2 + 5 \times 16^1 + 15 \times 16^{-0} = (46687)_{10}$$

- For our purposes, base-8 and base-16 are most useful as a “shorthand”
- notation for binary numbers

Octal (Hexadecimal) to Binary and Back

- Octal (Hexadecimal) to Binary:
 - Restate the octal (hexadecimal) as three (four) binary digits starting at the radix point and going both ways.
- Binary to Octal (Hexadecimal):
 - Group the binary digits into three (four) bit groups starting at the radix point and going both ways, padding with zeros as needed in the fractional part.
 - Convert each group of three bits to an octal (hexadecimal) digit.

Octal to Hexadecimal via Binary

- Convert octal to binary.
- Use groups of four bits and convert as above to hexadecimal digits.
- Example: Octal to Binary to Hexadecimal

6 3 5 . 1 7 7 8

- Why do these conversions work?

A Final Conversion Note

- You can use arithmetic in other bases if you are careful:
- Example: Convert 101110_2 to Base 10 using binary arithmetic:

Step 1 $101110 / 1010 = 100 \text{ r } 0110$

Step 2 $100 / 1010 = 0 \text{ r } 0100$

Converted Digits are $0100_2 | 0110_2$

or 4 6 10

Binary Numbers and Binary Coding

- Flexibility of representation
 - Within constraints below, can assign any binary combination (called a code word) to any data as long as data is uniquely encoded.
- Information Types
 - Numeric
 - Must represent range of data needed
 - Very desirable to represent data such that simple, straightforward computation for common arithmetic operations permitted
 - Tight relation to binary numbers
 - Non-numeric
 - Greater flexibility since arithmetic operations not applied.
 - Not tied to binary numbers

Non-numeric Binary Codes

- Given n binary digits (called bits), a binary code is a mapping from a set of represented elements to a subset of the 2^n binary numbers.
- Example: A binary code for the seven colors of the rainbow
- Code 100 is not used

Color	Binary Number
Red	000
Orange	001
Yellow	010
Green	011
Blue	100
Indigo	101
Violet	111

Number of Bits Required

- Given M elements to be represented by a binary code, the minimum number of bits, n , needed, satisfies the following relationships:

$$2^n \leq M < 2^{n+1}$$

$$\lceil n \rceil = \log_2 M$$

where $\lceil x \rceil$ called the *ceiling function*, which is the integer greater than or equal to x .

- Example: How many bits are required to represent decimal digits with a binary code?

Number of Elements Represented

- Given n digits in radix r , there are r^n distinct elements that can be represented.
- But, you can represent m elements, $m < r^n$
- Examples:
 - You can represent 4 elements in radix $r = 2$ with $n = 2$ digits: (00, 01, 10, 11).
 - You can represent 4 elements in radix $r = 2$ with $n = 4$ digits: (0001, 0010, 0100, 1000).
 - This second code is called a "one hot" code.

Binary Codes for Decimal Digits

- There are over 8,000 ways that you can chose 10 elements from the 16 binary numbers of 4 bits. A few are useful:

Decimal	8,4,2,1	Excess3	8,4,-2,-1	Gray
0	0000	0011	0000	0000
1	0001	0100	0111	0100
2	0010	0101	0110	0101
3	0011	0110	0101	0111
4	0100	0111	0100	0110
5	0101	1000	1011	0010
6	0110	1001	1010	0011
7	0111	1010	1001	0001
8	1000	1011	1000	1001
9	1001	1100	1111	1000

Binary Coded Decimal (BCD)

- The BCD code is the 8,4,2,1 code.
- This code is the simplest, most intuitive binary code for decimal digits and uses the same powers of 2 as a binary number, but only encodes the first ten values from 0 to 9.
- Example: $1001 \text{ (9)} = 1000 \text{ (8)} + 0001 \text{ (1)}$
- How many “invalid” code words are there?
- What are the “invalid” code words?

Excess 3 Code and 8, 4, -2, -1 Code

Decimal	Excess 3	8, 4, -2, -1
0	0011	0000
1	0100	0111
2	0101	0110
3	0110	0101
4	0111	0100
5	1000	1011
6	1001	1010
7	1010	1001
8	1011	1000
9	1100	1111

- What interesting property is common to these two codes?

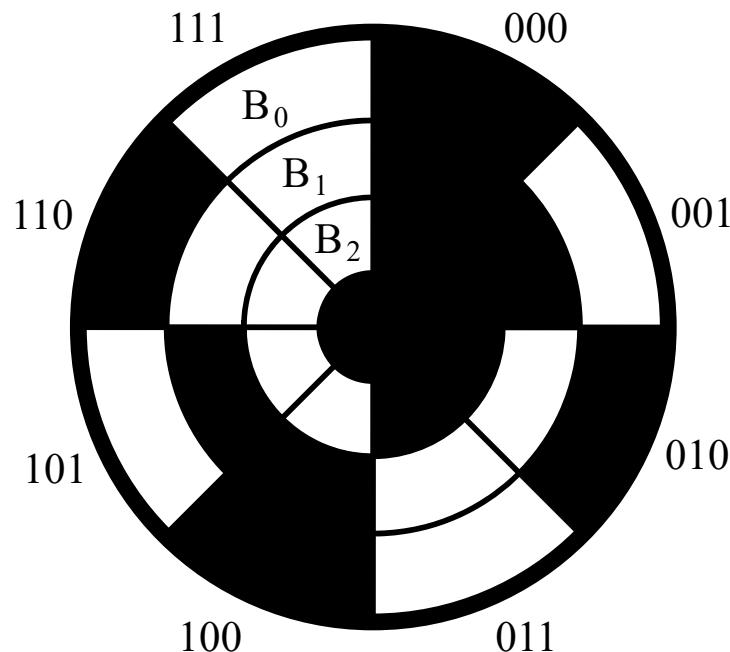
Gray Code

Decimal	8,4,2,1	Gray
0	0000	0000
1	0001	0100
2	0010	0101
3	0011	0111
4	0100	0110
5	0101	0010
6	0110	0011
7	0111	0001
8	1000	1001
9	1001	1000

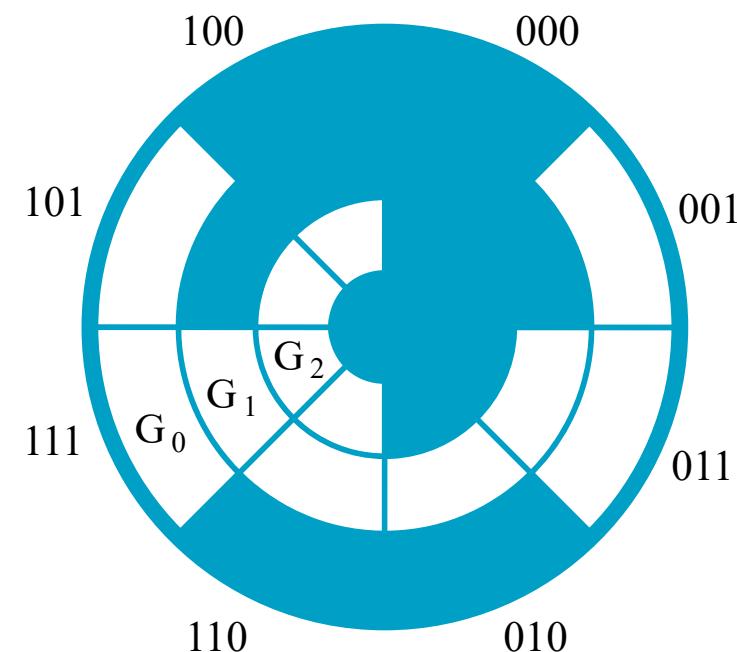
- What special property does the Gray code have in relation to adjacent decimal digits?

Gray Code (Continued)

- Does this special Gray code property have any value?
- An Example: Optical Shaft Encoder



(a) Binary Code for Positions 0 through 7



(b) Gray Code for Positions 0 through 7

Gray Code (Continued)

- How does the shaft encoder work?
- For the binary code, what codes may be produced if the shaft position lies between codes for 3 and 4 (011 and 100)?
- Is this a problem?

Gray Code (Continued)

- For the Gray code, what codes may be produced if the shaft position lies between codes for 3 and 4 (010 and 110)?
- Is this a problem?
- Does the Gray code function correctly for these borderline shaft positions for all cases encountered in octal counting?

Warning: Conversion or Coding?

- Do NOT mix up conversion of a decimal number to a binary number with coding a decimal number with a BINARY CODE.
- $13_{10} = 1101_2$ (This is conversion)
- $13 \Leftrightarrow 0001|0011$ (This is coding)

Chapter 2: Combinational Logic Circuits

Binary Logic

- Binary logic deals with binary variables, which take on two discrete values
- Associated with binary variables are three basic logical operations

Operation:

AND

OR

NOT

Expression:

xy or $x.y$

$x + y$

x'

Truth table:

x	y	xy
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$x+y$
0	0	0
0	1	1
1	0	1
1	1	1

x	x'
0	1
1	0

Logic Gates

- Each of basic operations can be implemented in hardware using a **logic gate**
 - Symbols for each of the logic gates are shown below
 - These gates output the AND, OR, and NOT of their inputs

Operation:

AND

OR

NOT

Expression:

xy or $x \cdot y$

$x + y$

x'

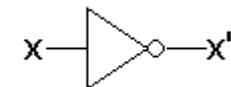
Logic gate:



AND gate



OR gate



NOT gate
(inverter)

Timing Diagram

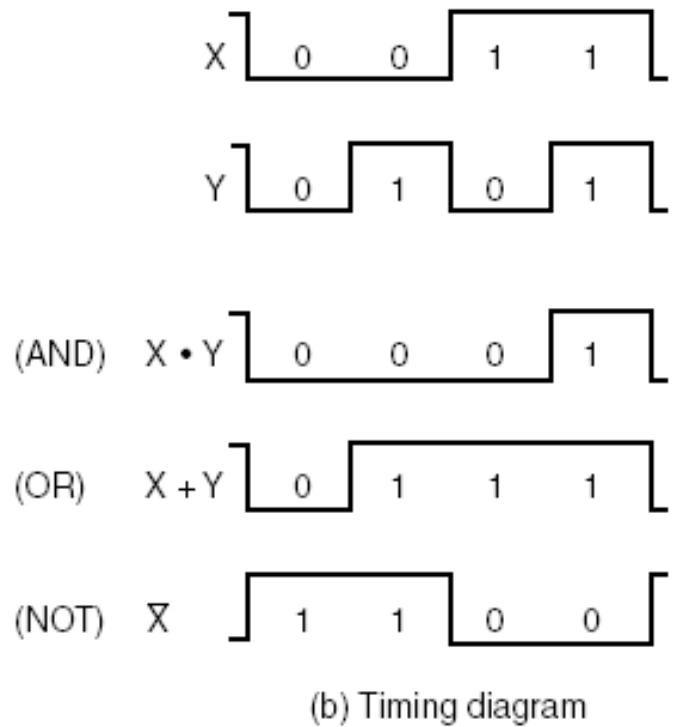


Fig. 2-1 Digital Logic Gates

Gates More Than Two Inputs

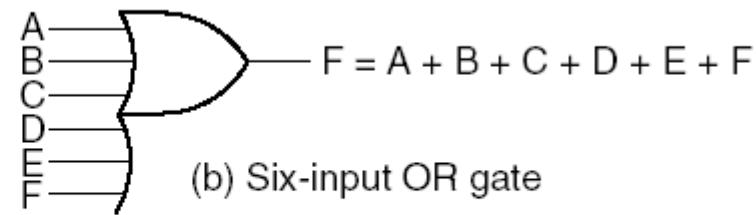
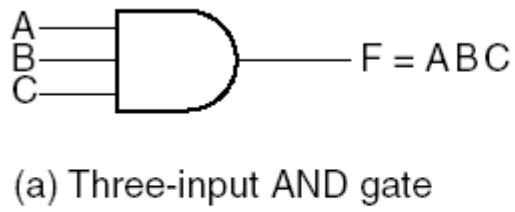


Fig. 2-2 Gates with More than Two Inputs

Boolean Function

- We can use basic operations to form complex Boolean functions, e.g.,

$$f(x,y,z) = (x + y')z + x'$$

- Some terminology and notation:

- f is the name of the function
 - (x,y,z) are the **input variables**, each representing 1 or 0. Listing the inputs is optional, but sometimes helpful
 - A **literal** is any occurrence of an input variable or its complement. The function above has four literals: x , y' , z , and x'
- Precedences are important, but not too difficult
 - NOT has the highest precedence, followed by AND, and then OR.
 - Fully parenthesized, the function above would be kind of messy:

$$f(x,y,z) = (((x + (y'))z) + x')$$

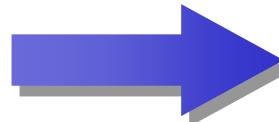
Truth Tables

- A **truth table** shows all possible inputs and outputs of a function
- Remember that each input variable represents either 1 or 0
 - Because there are only a finite number of values (1 and 0), truth tables themselves are finite
 - A function with n variables has 2^n possible combinations of inputs.
- Inputs are listed in binary order—in this example, from 000 to 111.

$$f(x,y,z) = (x + y')z + x'$$



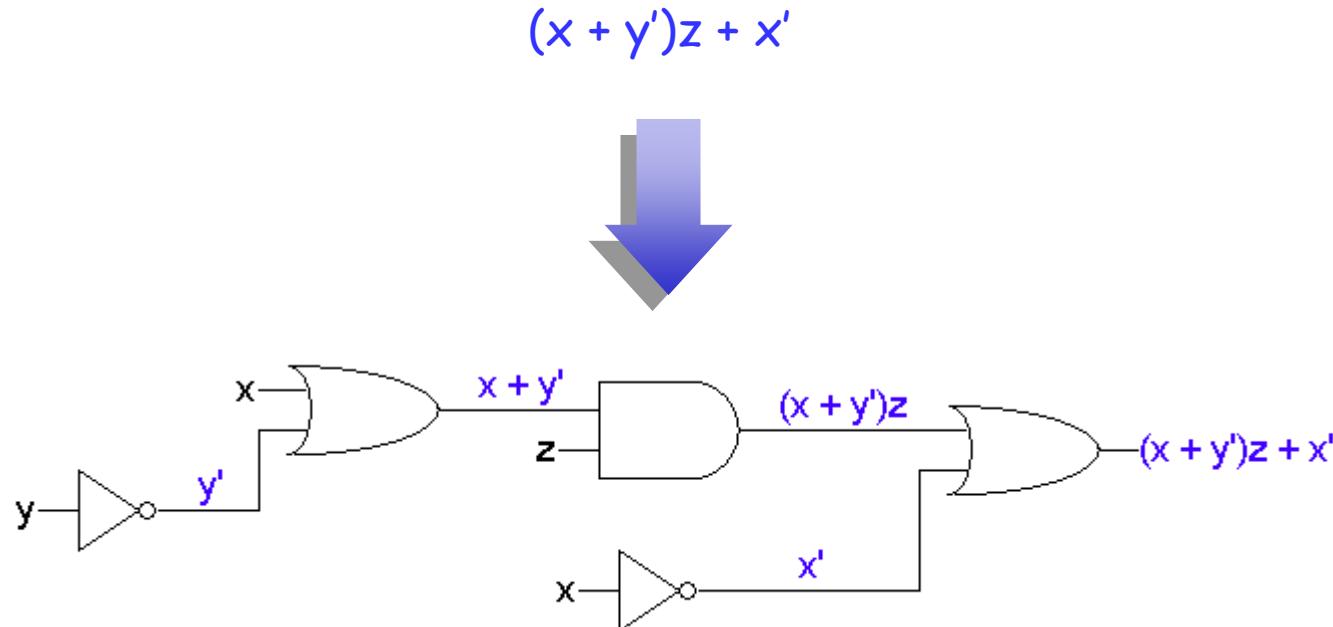
$$\begin{aligned} f(0,0,0) &= (0 + 1)0 + 1 = 1 \\ f(0,0,1) &= (0 + 1)1 + 1 = 1 \\ f(0,1,0) &= (0 + 0)0 + 1 = 1 \\ f(0,1,1) &= (0 + 0)1 + 1 = 1 \\ f(1,0,0) &= (1 + 1)0 + 0 = 0 \\ f(1,0,1) &= (1 + 1)1 + 0 = 1 \\ f(1,1,0) &= (1 + 0)0 + 0 = 0 \\ f(1,1,1) &= (1 + 0)1 + 0 = 1 \end{aligned}$$



x	y	z	f(x,y,z)
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Expressions and Circuits

- Any Boolean expression can be converted into a **circuit** by combining basic gates
- The diagram below shows the inputs and outputs of each gate
- The precedences are explicit in a circuit. Clearly, we have to make sure that the hardware does operations in the right order!



Boolean Algebra

- A Boolean algebra requires
 - A set of elements B , which needs *at least* two elements (0 and 1)
 - Two binary (two-argument) operations OR and AND
 - A unary (one-argument) operation NOT
 - The **axioms** below must always be true
 - The **magenta axioms** deal with the complement operation
 - **Blue axioms** (especially 15) are different from regular algebra

$$1. \quad x + 0 = x$$

$$2. \quad x \cdot 1 = x$$

$$3. \quad x + 1 = 1$$

$$4. \quad x \cdot 0 = 0$$

$$5. \quad x + x = x$$

$$6. \quad x \cdot x = x$$

$$7. \quad x + x' = 1$$

$$8. \quad x \cdot x' = 0$$

$$9. \quad (x')' = x$$

$$10. \quad x + y = y + x$$

$$11. \quad xy = yx$$

Commutative

$$12. \quad x + (y + z) = (x + y) + z$$

$$13. \quad x(yz) = (xy)z$$

Associative

$$14. \quad x(y + z) = xy + xz$$

$$15. \quad x + yz = (x + y)(x + z)$$

Distributive

$$16. \quad (x + y)' = x'y'$$

$$17. \quad (xy)' = x' + y'$$

DeMorgan's

Comments on the Axioms

- The left and right columns of axioms are **DUALS**
 - exchange all ANDs with ORs, and 0s with 1s
- The duality principle of Boolean algebra: A Boolean equation remains valid if we take the dual of the expression on both side of the equal signs

$$1. \quad x + 0 = x$$

$$2. \quad x \cdot 1 = x$$

$$3. \quad x + 1 = 1$$

$$4. \quad x \cdot 0 = 0$$

$$5. \quad x + x = x$$

$$6. \quad x \cdot x = x$$

$$7. \quad x + x' = 1$$

$$8. \quad x \cdot x' = 0$$

$$9. \quad (x')' = x$$

$$10. \quad x + y = y + x$$

$$11. \quad xy = yx$$

Commutative

$$12. \quad x + (y + z) = (x + y) + z$$

$$13. \quad x(yz) = (xy)z$$

Associative

$$14. \quad x(y + z) = xy + xz$$

$$15. \quad x + yz = (x + y)(x + z)$$

Distributive

$$16. \quad (x + y)' = x'y'$$

$$17. \quad (xy)' = x' + y'$$

DeMorgan's

Are these axioms for real?

- We can show that these axioms are true, given the definitions of AND, OR and NOT

x	y	xy
0	0	0
0	1	0
1	0	0
1	1	1

x	y	x+y
0	0	0
0	1	1
1	0	1
1	1	1

x	x'
0	1
1	0

- The first 11 axioms are easy to see from these truth tables alone. For example, $x + x' = 1$ because of the middle two lines below (where $y = x'$)

x	y	x+y
0	0	0
0	1	1
1	0	1
1	1	1

Is $X+YZ = (X+Y)(X+Z)$?

X	Y	Z	X+Y	X+Z	YZ	X+YZ	(X+Y)(X+Z)
0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	1	1	1	1
1	0	0	1	1	0	1	1
1	0	1	1	1	0	1	1
1	1	0	1	1	0	1	1
1	1	1	1	1	1	1	1

DeMorgan's Theorem

- We can make up truth tables to prove (both parts of) DeMorgan's law
- For $(x + y)' = x'y'$, we can make truth tables for $(x + y)'$ and for $x'y'$

x	y	$x + y$	$(x + y)'$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	0

x	y	x'	y'	$x'y'$
0	0	1	1	1
0	1	1	0	0
1	0	0	1	0
1	1	0	0	0

- Since both of the columns in blue are the same, this shows that $(x + y)'$ and $x'y'$ are equivalent

$$\overline{X_1 + X_2 + \dots + X_n} = \overline{X}_1 \overline{X}_2 \dots \overline{X}_n$$

$$\overline{X_1 \cdot X_2 \dots X_n} = \overline{X}_1 + \overline{X}_2 + \dots + \overline{X}_n$$

Simplification with Axioms

- We can now start doing some simplifications

$$\begin{aligned} & x'y' + xyz + x'y \\ &= x'(y' + y) + xyz \quad [\text{Distributive}; x'y' + x'y = x'(y' + y)] \\ &= x' \bullet 1 + xyz \quad [\text{Axiom 7}; y' + y = 1] \\ &= x' + xyz \quad [\text{Axiom 2}; x' \bullet 1 = x'] \\ &= (x' + x)(x' + yz) \quad [\text{Distributive}] \\ &= 1 \bullet (x' + yz) \quad [\text{Axiom 7}; x' + x = 1] \\ &= x' + yz \quad [\text{Axiom 2}] \end{aligned}$$

$$1. x + 0 = x$$

$$3. x + 1 = 1$$

$$5. x + x = x$$

$$7. x + x' = 1$$

$$9. (x')' = x$$

$$2. x \bullet 1 = x$$

$$4. x \bullet 0 = 0$$

$$6. x \bullet x = x$$

$$8. x \bullet x' = 0$$

$$10. x + y = y + x \qquad \qquad \qquad 11. xy = yx \qquad \qquad \qquad \text{Commutative}$$

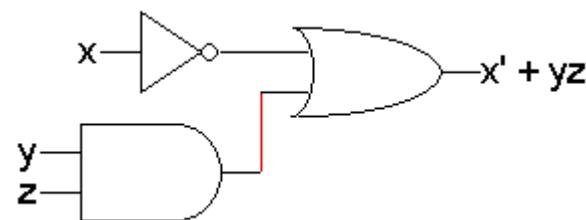
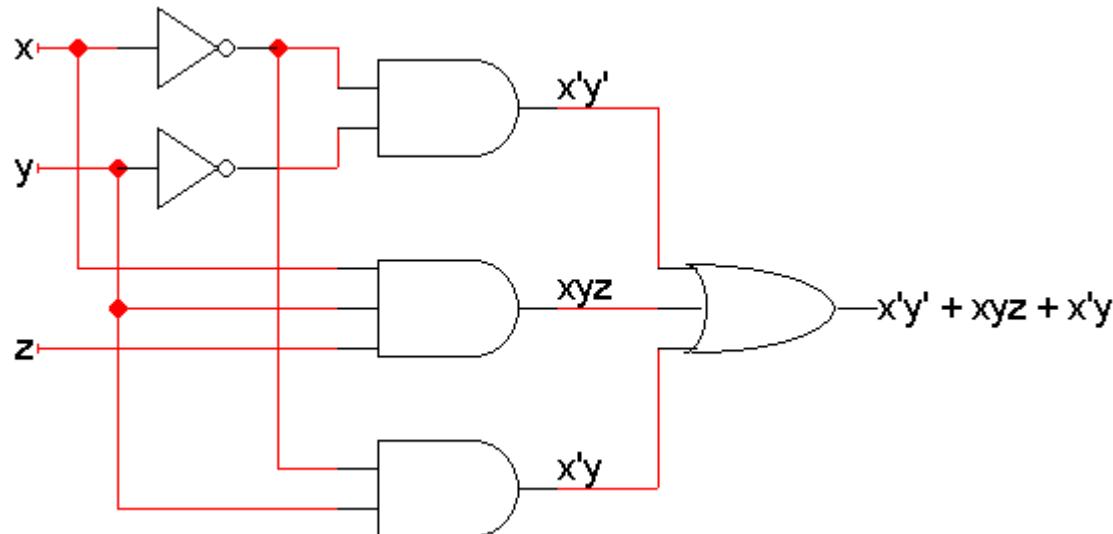
$$12. x + (y + z) = (x + y) + z \qquad 13. x(yz) = (xy)z \qquad \qquad \qquad \text{Associative}$$

$$14. x(y + z) = xy + xz \qquad 15. x + yz = (x + y)(x + z) \qquad \qquad \qquad \text{Distributive}$$

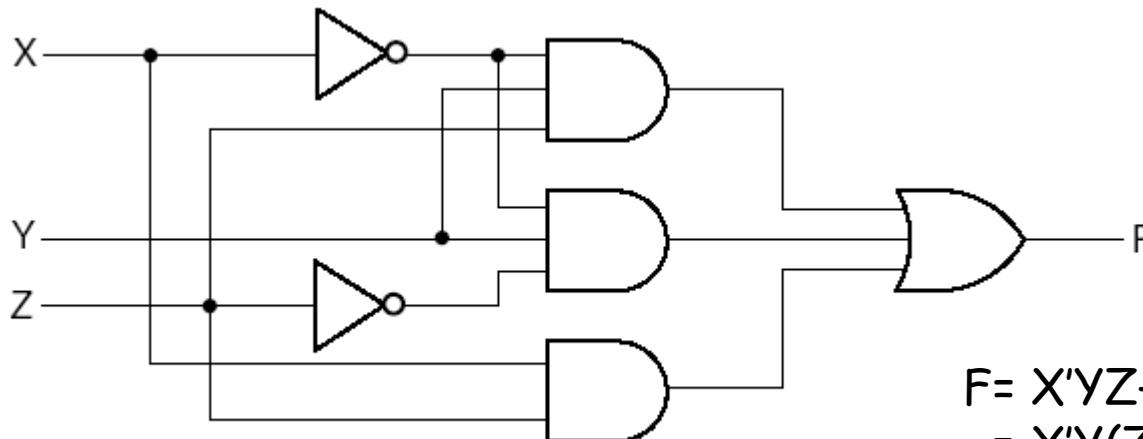
$$16. (x + y)' = x'y' \qquad \qquad \qquad 17. (xy)' = x' + y' \qquad \qquad \qquad \text{DeMorgan's}$$

Let's compare the resulting circuits

- Here are two different but *equivalent* circuits
- In general the one with fewer gates is "better":
 - It costs less to build
 - It requires less power
 - But we had to do some work to find the second form

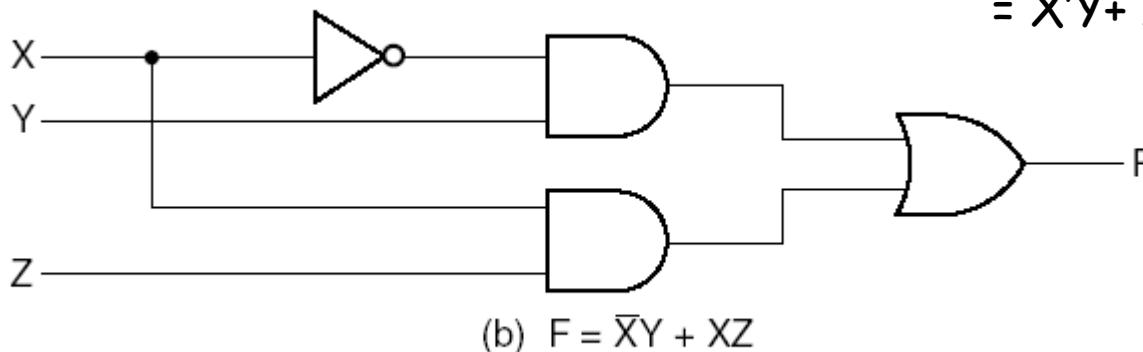


Another Example



$$(a) F = \bar{X}YZ + \bar{X}Y\bar{Z} + XZ$$

$$\begin{aligned} F &= X'YZ + X'YZ' + XZ \\ &= X'Y(Z+Z') + XZ \quad (14) \\ &= X'Y \cdot 1 + XZ \quad (7) \\ &= X'Y + XZ \quad (2) \end{aligned}$$



$$(b) F = \bar{X}Y + XZ$$

Fig. 2-4 Implementation of Boolean Function with Gates

Some More Laws

- Here are some more useful laws. Notice the duals again

$$1. \quad x + xy = x$$

$$2. \quad xy + xy' = x$$

$$3. \quad x + x'y = x + y$$

$$4. \quad x(x + y) = x$$

$$5. \quad (x + y)(x + y') = x$$

$$6. \quad x(x' + y) = xy$$

Consensus Theorem

- $XY + X'Z + YZ = XY + X'Z$ or its dual

$$(X+Y)(X'+Z)(Y+Z) = (X+Y)(X'+Z)$$

- The proof of the theorem:

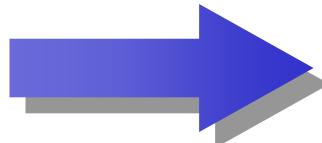
$$\begin{aligned} XY + X'Z + YZ &= XY + X'Z + YZ(X + X') \\ &= XY + X'Z + XYZ + X'YZ \\ &= XY + XYZ + X'Z + X'YZ \\ &= XY(1 + Z) + X'Z(1 + Y) \\ &= XY + X'Z \end{aligned}$$

The Complement of a Function

- The complement of a function always outputs 0 where the original function output 1, and 1 where the original produced 0
- In a truth table, we can just exchange 0s and 1s in the output column(s)

$$f(x,y,z) = x' + xyz'$$

x	y	z	f(x,y,z)
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0



x	y	z	f'(x,y,z)
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Complementing a Function Algebraically

- You can use DeMorgan's law to keep "pushing" the complements inwards

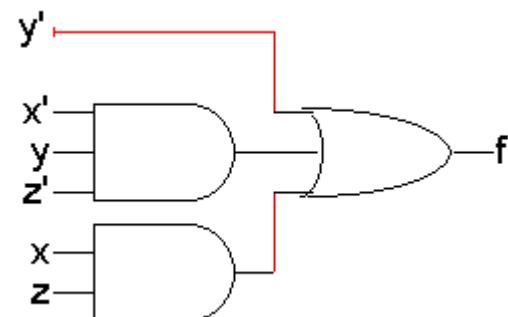
$$f(x,y,z) = x(y'z' + yz)$$

$$\begin{aligned} f'(x,y,z) &= (x(y'z' + yz))' && [\text{complement both sides}] \\ &= x' + (y'z' + yz)' && [\text{because } (xy)' = x' + y'] \\ &= x' + (y'z')' (yz)' && [\text{because } (x + y)' = x' y'] \\ &= x' + (y + z)(y' + z') && [\text{because } (xy)' = x' + y', \text{ twice}] \end{aligned}$$

- You can also take the dual of the function, and then complement each literal
 - If $f(x,y,z) = x(y'z' + yz)$...
 - ...the dual of f is $x + (y' + z')(y + z)$...
 - ...then complementing each literal gives $x' + (y + z)(y' + z')$...
 - ...so $f'(x,y,z) = x' + (y + z)(y' + z')$

Standard Forms of Expression

- We can write expressions in many ways, but some ways are more useful than others
- A **sum of products (SOP)** expression contains:
 - Only OR (sum) operations at the "outermost" level
 - Each term that is summed must be a product of literals
$$f(x,y,z) = y' + x'yz' + xz$$
- The advantage is that any sum of products expression can be implemented using a **two-level circuit**
 - literals and their complements at the "0th" level
 - AND gates at the first level
 - a single OR gate at the second level
- This diagram uses some shorthands...
 - NOT gates are implicit
 - literals are reused



Minterms

- A **minterm** is a special product of literals, in which each input variable appears exactly once
- A function with n variables has 2^n minterms (since each variable can appear complemented or not)
- A three-variable function, such as $f(x,y,z)$, has $2^3 = 8$ minterms:
$$\begin{array}{cccc} x'y'z' & x'y'z & x'y'z & x'yz \\ xy'z' & xy'z & xyz' & xyz \end{array}$$
- Each minterm is true for exactly one combination of inputs:

Minterm	Is true when...	Shorthand
$x'y'z'$	$x=0, y=0, z=0$	m_0
$x'y'z$	$x=0, y=0, z=1$	m_1
$x'yz'$	$x=0, y=1, z=0$	m_2
$x'yz$	$x=0, y=1, z=1$	m_3
$xy'z'$	$x=1, y=0, z=0$	m_4
$xy'z$	$x=1, y=0, z=1$	m_5
xyz'	$x=1, y=1, z=0$	m_6
xyz	$x=1, y=1, z=1$	m_7

Sum of Minterms Form

- Every function can be written as a **sum of minterms**, which is a special kind of sum of products form
- The sum of minterms form for any function is *unique*
- If you have a truth table for a function, you can write a sum of minterms expression just by picking out the rows of the table where the function output is 1.

x	y	z	f(x,y,z)	f'(x,y,z)
0	0	0	1	0
0	0	1	1	0
0	1	0	1	0
0	1	1	1	0
1	0	0	0	1
1	0	1	0	1
1	1	0	1	0
1	1	1	0	1

$$\begin{aligned}f &= x'y'z' + x'y'z + x'yz' + x'yz + xyz' \\&= m_0 + m_1 + m_2 + m_3 + m_6 \\&= \Sigma m(0,1,2,3,6)\end{aligned}$$

$$\begin{aligned}f' &= xy'z' + xy'z + xyz \\&= m_4 + m_5 + m_7 \\&= \Sigma m(4,5,7)\end{aligned}$$

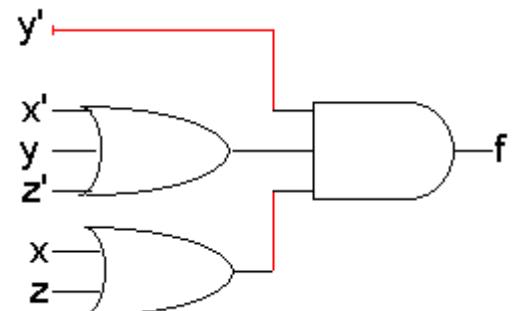
f' contains all the minterms not in f

The Dual Idea: Products of Sums

- A **product of sums (POS)** expression contains:
 - Only AND (product) operations at the “outermost” level
 - Each term must be a sum of literals

$$f(x,y,z) = y' (x' + y + z') (x + z)$$

- Product of sums expressions can be implemented with two-level circuits
 - literals and their complements at the “0th” level
 - *OR gates* at the first level
 - a single *AND gate* at the second level
- Compare this with sums of products



Maxterms

- A **maxterm** is a *sum* of literals, in which each input variable appears exactly once
- A function with n variables has 2^n maxterms
- The maxterms for a three-variable function $f(x,y,z)$:
$$\begin{array}{cccc} x' + y' + z' & x' + y' + z & x' + y + z' & x' + y + z \\ x + y' + z' & x + y' + z & x + y + z' & x + y + z \end{array}$$
- Each maxterm is *false* for exactly one combination of inputs:

Maxterm	Is <i>false</i> when...	Shorthand
$x + y + z$	$x=0, y=0, z=0$	M_0
$x + y + z'$	$x=0, y=0, z=1$	M_1
$x + y' + z$	$x=0, y=1, z=0$	M_2
$x + y' + z'$	$x=0, y=1, z=1$	M_3
$x' + y + z$	$x=1, y=0, z=0$	M_4
$x' + y + z'$	$x=1, y=0, z=1$	M_5
$x' + y' + z$	$x=1, y=1, z=0$	M_6
$x' + y' + z'$	$x=1, y=1, z=1$	M_7

Product of Maxterms Form

- Every function can be written as a *unique product of maxterms*
- If you have a truth table for a function, you can write a product of maxterms expression by picking out the rows of the table where the function output is 0. (Be careful if you're writing the actual literals!)

x	y	z	f(x,y,z)	f'(x,y,z)
0	0	0	1	0
0	0	1	1	0
0	1	0	1	0
0	1	1	1	0
1	0	0	0	1
1	0	1	0	1
1	1	0	1	0
1	1	1	0	1

$$\begin{aligned}f &= (x' + y + z)(x' + y + z')(x' + y' + z') \\&= M_4 M_5 M_7 \\&= \prod M(4,5,7)\end{aligned}$$

$$\begin{aligned}f' &= (x + y + z)(x + y + z')(x + y' + z) \\&\quad (x + y' + z')(x' + y' + z) \\&= M_0 M_1 M_2 M_3 M_6 \\&= \prod M(0,1,2,3,6)\end{aligned}$$

f' contains all the maxterms not in f

Minterms and maxterms are related

- Any minterm m_i is the *complement* of the corresponding maxterm M_i

Minterm	Shorthand	Maxterm	Shorthand
$x'y'z'$	m_0	$x + y + z$	M_0
$x'y'z$	m_1	$x + y + z'$	M_1
$x'yz'$	m_2	$x + y' + z$	M_2
$x'yz$	m_3	$x + y' + z'$	M_3
$xy'z'$	m_4	$x' + y + z$	M_4
$xy'z$	m_5	$x' + y + z'$	M_5
xyz'	m_6	$x' + y' + z$	M_6
xyz	m_7	$x' + y' + z'$	M_7

- For example, $m_4' = M_4$ because $(xy'z')' = x' + y + z$

Converting Between Standard Forms

- We can convert a sum of minterms to a product of maxterms

From before $f = \Sigma m(0,1,2,3,6)$
and $f' = \Sigma m(4,5,7)$
 $= m_4 + m_5 + m_7$

complementing $(f')' = (m_4 + m_5 + m_7)'$
so $f = m_4' m_5' m_7' \quad [\text{DeMorgan's law}]$
 $= M_4 M_5 M_7 \quad [\text{By the previous page}]$
 $= \prod M(4,5,7)$

- In general, just replace the minterms with maxterms, using maxterm numbers that don't appear in the sum of minterms:

$$\begin{aligned} f &= \Sigma m(0,1,2,3,6) \\ &= \prod M(4,5,7) \end{aligned}$$

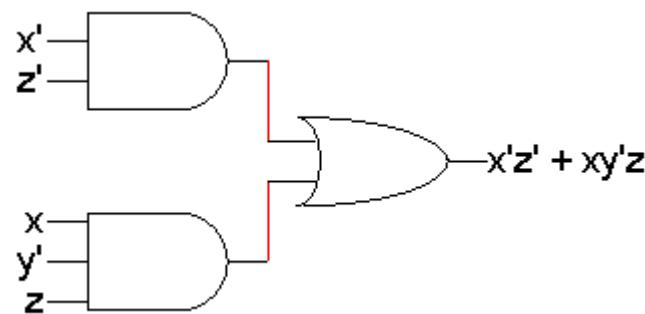
- The same thing works for converting from a product of maxterms to a sum of minterms

Summary

- So far:
 - A bunch of Boolean algebra trickery for simplifying expressions and circuits
 - The algebra guarantees us that the simplified circuit is *equivalent* to the original one
 - Introducing some standard forms and terminology
- Next:
 - An alternative simplification method
 - We'll start using all this stuff to build and analyze bigger, more useful, circuits

Karnaugh Maps

- Applications of Boolean logic to circuit design
 - The basic Boolean operations are AND, OR and NOT
 - These operations can be combined to form complex expressions, which can also be directly translated into a hardware circuit
 - Boolean algebra helps us simplify expressions and circuits
- Karnaugh Map: A graphical technique for simplifying an expression into a **minimal sum of products (MSP)** form:
 - There are a minimal number of product terms in the expression
 - Each term has a minimal number of literals
- Circuit-wise, this leads to a *minimal/two-level implementation*



Review: Minterm

- A **product** term in which all the variables appear exactly once, either complemented or uncomplemented, is called a **minterm**
- A minterm represents exactly one combination of the binary variables in a truth table. It has the value of 1 for that combination and 0 for the others

X	Y	Z	Product Term	Symbol	m_0	m_1	m_2	m_3	m_4	m_5	m_6	m_7
0	0	0	$\overline{X}\overline{Y}\overline{Z}$	m_0	1	0	0	0	0	0	0	0
0	0	1	$\overline{X}\overline{Y}Z$	m_1	0	1	0	0	0	0	0	0
0	1	0	$\overline{X}YZ$	m_2	0	0	1	0	0	0	0	0
0	1	1	$\overline{X}Y\overline{Z}$	m_3	0	0	0	1	0	0	0	0
1	0	0	$X\overline{Y}\overline{Z}$	m_4	0	0	0	0	1	0	0	0
1	0	1	$X\overline{Y}Z$	m_5	0	0	0	0	0	1	0	0
1	1	0	$X\overline{Y}\overline{Z}$	m_6	0	0	0	0	0	0	1	0
1	1	1	XYZ	m_7	0	0	0	0	0	0	0	1

Table 2-6 Minterms for Three Variables

Review: Maxterm

- A sum term in which all the variables appear exactly once, either complemented or uncomplemented, is called a maxterm
- A maxterm represents exactly one combination of the binary variables in a truth table. It has the value of 0 for that combination and 1 for the others

X	Y	Z	Sum Term	Symbol	M ₀	M ₁	M ₂	M ₃	M ₄	M ₅	M ₆	M ₇
0	0	0	$X + Y + Z$	M ₀	0	1	1	1	1	1	1	1
0	0	1	$X + Y + \bar{Z}$	M ₁	1	0	1	1	1	1	1	1
0	1	0	$X + \bar{Y} + Z$	M ₂	1	1	0	1	1	1	1	1
0	1	1	$X + \bar{Y} + \bar{Z}$	M ₃	1	1	1	0	1	1	1	1
1	0	0	$\bar{X} + Y + Z$	M ₄	1	1	1	1	0	1	1	1
1	0	1	$\bar{X} + Y + \bar{Z}$	M ₅	1	1	1	1	1	0	1	1
1	1	0	$\bar{X} + \bar{Y} + Z$	M ₆	1	1	1	1	1	1	0	1
1	1	1	$\bar{X} + \bar{Y} + \bar{Z}$	M ₇	1	1	1	1	1	1	1	0

Table 2-7 Maxterms for Three Variables

- A minterm and maxterm with the same subscript are the complements of each other, i.e., $M_j = m'_j$

Review: Sum of Minterms

- A Boolean function can be represented algebraically from a given truth table by forming the logical sum of all the minterms that produce a 1 in the function. This expression is called a **sum of minterms**

(a)	X	Y	Z	F	\bar{F}
	0	0	0	1	0
	0	0	1	0	1
	0	1	0	1	0
	0	1	1	0	1
	1	0	0	0	1
	1	0	1	1	0
	1	1	0	0	1
	1	1	1	1	0

$$\begin{aligned} F &= X'Y'Z' + X'YZ' + XY'Z + XYZ \\ &= m_0 + m_2 + m_5 + m_7 \end{aligned}$$

$$F(X,Y,Z) = \Sigma m(0,2,5,7)$$

Review: Product of Maxterms

- A Boolean function can be represented algebraically from a given truth table by forming the logical product of all the maxterms that produce a 0 in the function. This expression is called a **product of maxterms**

(a)	X	Y	Z	F	\bar{F}
0	0	0	0	1	0
0	0	0	1	0	1
0	1	0	0	1	0
0	1	0	1	0	1
1	0	0	0	0	1
1	0	0	1	1	0
1	0	1	0	1	0
1	0	1	1	0	1
1	1	0	0	1	0
1	1	0	1	0	1
1	1	1	0	0	1
1	1	1	1	1	0

$$\begin{aligned} F &= (X+Y+Z')(X+Y'+Z')(X'+Y+Z)(X'+Y'+Z) \\ &= M_1 \cdot M_3 \cdot M_4 \cdot M_6 \end{aligned}$$

$$F(X,Y,Z) = \prod M(1,3,4,6)$$

- To convert a Boolean function F from SoM to PoM:
 - Find F' in SoM form
 - Find $F = (F')'$ in PoM form

Review: Important Properties of Minterms

- There are 2^n minterms for n Boolean variables. These minterms can be evaluated from the binary numbers from 0 to $2^n - 1$
- Any Boolean function can be expressed as a logical sum of minterms
- The complement of a function contains those minterms not included in the original function

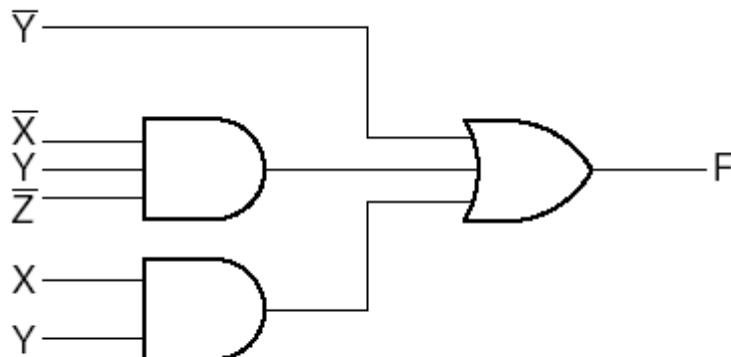
$$F(X,Y,Z) = \Sigma m(0,2,5,7) \Rightarrow F'(X,Y,Z) = \Sigma m(1,3,4,6)$$

- A function that includes all the 2^n minterms is equal to logic 1

$$G(X,Y) = \Sigma m(0,1,2,3) = 1$$

Review: Sum-of-Products

- The sum-of-minterms form is a standard algebraic expression that is obtained from a truth table
- When we simplify a function in SoM form by reducing the number of product terms or by reducing the number of literals in the terms, the simplified expression is said to be in **Sum-of-Products** form
- Sum-of-Products expression can be implemented using a **two-level circuit**

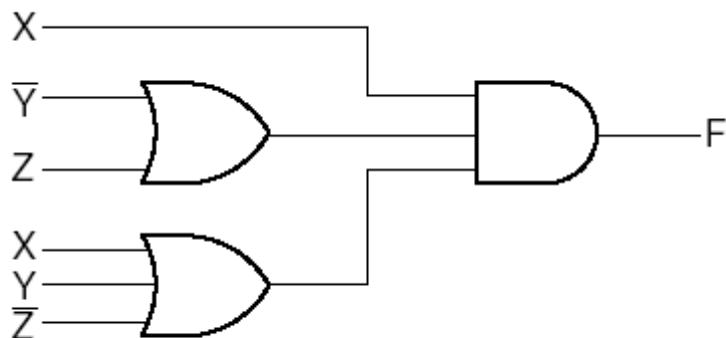


$$\begin{aligned} F &= \Sigma m(0,1,2,3,4,5,7) && (\text{SoM}) \\ &= Y' + X'YZ' + XY && (\text{SoP}) \end{aligned}$$

Fig. 2-5 Sum-of-Products Implementation

Review: Product-of-Sums

- The product-of-maxterms form is a standard algebraic expression that is obtained from a truth table
- When we simplify a function in PoM form by reducing the number of sum terms or by reducing the number of literals in the terms, the simplified expression is said to be in **Product-of-Sums** form
- Product-of-Sums expression can be implemented using a two-level circuit



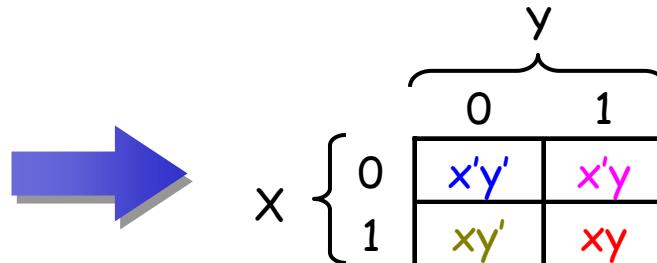
$$\begin{aligned} F &= \prod M(0,2,3,4,5,6) && (\text{PoM}) \\ &= X(Y' + Z)(X + Y + Z') && (\text{PoS}) \end{aligned}$$

Fig. 2-7 Product-of-Sums Implementation

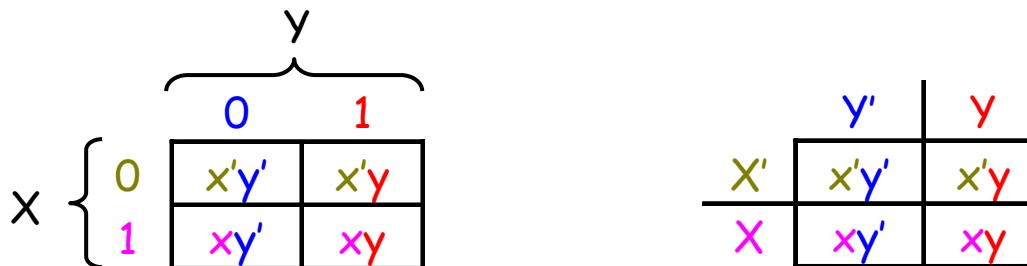
Re-arranging the Truth Table

- A two-variable function has four possible minterms. We can re-arrange these minterms into a **Karnaugh map**

x	y	minterm
0	0	$x'y'$
0	1	$x'y$
1	0	xy'
1	1	xy



- Now we can easily see which minterms contain common literals
 - Minterms on the left and right sides contain y' and y respectively
 - Minterms in the top and bottom rows contain x' and x respectively



Karnaugh Map Simplifications

- Imagine a two-variable sum of minterms:

$$x'y' + x'y$$

- Both of these minterms appear in the top row of a Karnaugh map, which means that they both contain the literal x'

		y
	$x'y'$	$x'y$
x	xy'	xy

- What happens if you simplify this expression using Boolean algebra?

$$\begin{aligned} x'y' + x'y &= x'(y' + y) && [\text{Distributive}] \\ &= x' \bullet 1 && [y + y' = 1] \\ &= x' && [x \bullet 1 = x] \end{aligned}$$

More Two-Variable Examples

- Another example expression is $x'y + xy$
 - Both minterms appear in the right side, where y is uncomplemented
 - Thus, we can reduce $x'y + xy$ to just y

		y
	$x'y'$	$x'y$
X	xy'	xy

- How about $x'y' + x'y + xy$?
 - We have $x'y' + x'y$ in the top row, corresponding to x'
 - There's also $x'y + xy$ in the right side, corresponding to y
 - This whole expression can be reduced to $x' + y$

		y
	$x'y'$	$x'y$
X	xy'	xy

A Three-Variable Karnaugh Map

- For a three-variable expression with inputs x, y, z , the arrangement of minterms is more tricky:

		YZ				
		00	01	11	10	
X		0	$x'y'z'$	$x'y'z$	$x'yz$	$x'yz'$
X		1	$xy'z'$	$xy'z$	xyz	xyz'

		YZ				
		00	01	11	10	
X		0	m_0	m_1	m_3	m_2
X		1	m_4	m_5	m_7	m_6

- Another way to label the K-map (use whichever you like):

		y			
		$x'y'z'$	$x'y'z$	$x'yz$	$x'yz'$
X		$xy'z'$	$xy'z$	xyz	xyz'
		Z			

		y			
		m_0	m_1	m_3	m_2
X		m_4	m_5	m_7	m_6
		Z			

Why the funny ordering?

- With this ordering, any group of 2, 4 or 8 adjacent squares on the map contains common literals that can be factored out

		y	
x	$x'y'z'$	$x'y'z$	$x'yz$
	$xy'z'$	$xy'z$	xyz

Z

$$\begin{aligned} & x'y'z + x'yz \\ = & x'z(y' + y) \\ = & x'z \bullet 1 \\ = & x'z \end{aligned}$$

- "Adjacency" includes wrapping around the left and right sides:

		y	
x	$x'y'z'$	$x'y'z$	$x'yz$
	$xy'z'$	$xy'z$	xyz

Z

$$\begin{aligned} & x'y'z' + xy'z' + x'yz' + xyz' \\ = & z'(x'y' + xy' + x'y + xy) \\ = & z'(y'(x' + x) + y(x' + x)) \\ = & z'(y'+y) \\ = & z' \end{aligned}$$

- We'll use this property of adjacent squares to do our simplifications.

Example K-map Simplification

- Let's consider simplifying $f(x,y,z) = xy + y'z + xz$
- First, you should convert the expression into a sum of minterms form, if it's not already
 - The easiest way to do this is to make a truth table for the function, and then read off the minterms
 - You can either write out the literals or use the minterm shorthand
- Here is the truth table and sum of minterms for our example:

x	y	z	$f(x,y,z)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

$$\begin{aligned}f(x,y,z) &= x'y'z + xy'z + xyz' + xyz \\&= m_1 + m_5 + m_6 + m_7\end{aligned}$$

Unsimplifying Expressions

- You can also convert the expression to a sum of minterms with Boolean algebra
 - Apply the distributive law in reverse to add in missing variables.
 - Very few people actually do this, but it's occasionally useful.

$$\begin{aligned}xy + y'z + xz &= (xy \bullet 1) + (y'z \bullet 1) + (xz \bullet 1) \\&= (xy \bullet (z' + z)) + (y'z \bullet (x' + x)) + (xz \bullet (y' + y)) \\&= (xyz' + xyz) + (x'y'z + xy'z) + (xy'z + xyz) \\&= \textcolor{blue}{xyz' + xyz + x'y'z + xy'z}\end{aligned}$$

- In both cases, we're actually "unsimplifying" our example expression
 - The resulting expression is larger than the original one!
 - But having all the individual minterms makes it easy to combine them together with the K-map

Making the Example K-map

- Next up is drawing and filling in the K-map
 - Put 1s in the map for each minterm, and 0s in the other squares
 - You can use either the minterm products or the shorthand to show you where the 1s and 0s belong
- In our example, we can write $f(x,y,z)$ in two equivalent ways

$$f(x,y,z) = x'y'z' + xy'z + xyz' + xyz$$

$$f(x,y,z) = m_1 + m_5 + m_6 + m_7$$

		y	
		x'y'z'	
x		x'y'z'	xy'z
		x'y'z	xyz'
x		xyz'	xyz

Z

		y	
		m ₀	m ₁
x		m ₄	m ₅
		m ₃	m ₂
x		m ₇	m ₆

Z

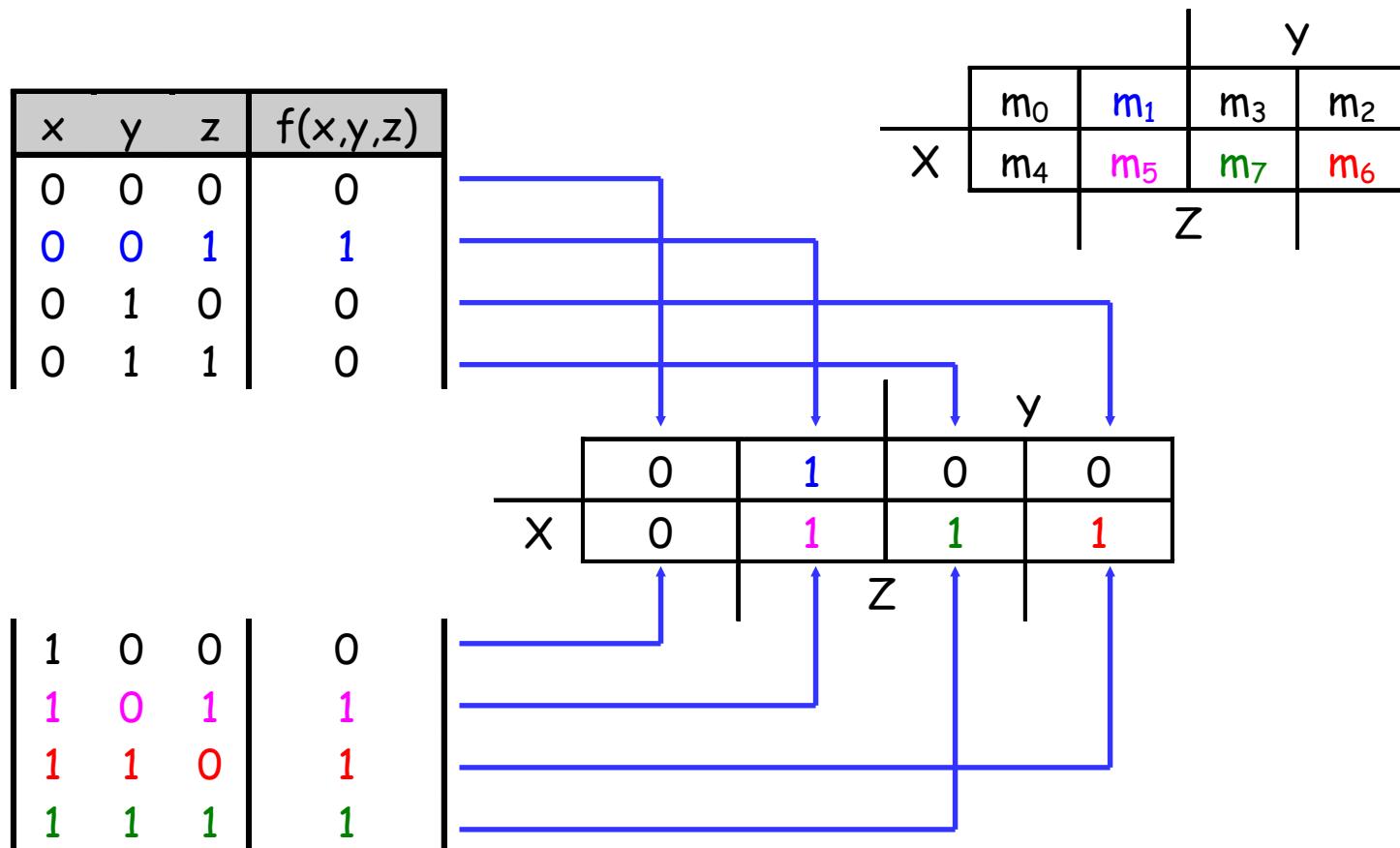
- In either case, the resulting K-map is shown below

		y	
		0	1
x		0	1
		0	1
x		1	0

Z

K-maps From Truth Tables

- You can also fill in the K-map directly from a truth table
 - The output in row i of the table goes into square m_i of the K-map
 - Remember that the rightmost columns of the K-map are "switched"



Grouping the Minterms Together

- The most difficult step is grouping together all the 1s in the K-map
 - Make **rectangles** around groups of one, two, four or eight 1s
 - All of the 1s in the map should be included in at least one rectangle
 - Do *not* include any of the 0s

		y	
	x	0	1
x	0	0	1
		0	0

- Each group corresponds to one product term. For the simplest result:
 - Make as few rectangles as possible, to minimize the number of products in the final expression.
 - Make each rectangle as large as possible, to minimize the number of literals in each term.
 - It's all right for rectangles to overlap, if that makes them larger.

Reading the MSP from the K-map

- Finally, you can find the minimal SoP expression
 - Each rectangle corresponds to one product term
 - The product is determined by finding the common literals in that rectangle

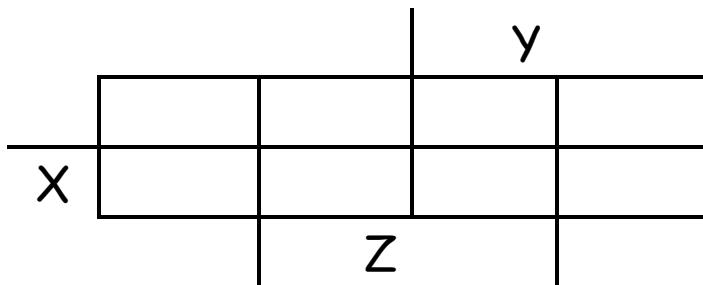
			y
	0	1	0
x	0	1	1
		z	

			y
	$x'y'z'$	$x'y'z$	$x'yz$
x	$xy'z'$	$xy'z$	xyz
		z	

- For our example, we find that $xy + y'z + xz = y'z + xy$. (This is one of the additional algebraic laws from last time.)

Practice K-map 1

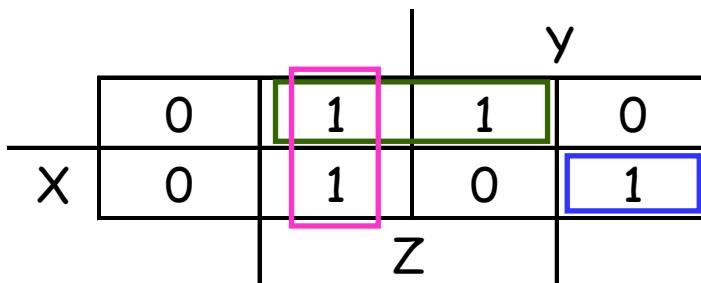
- Simplify the sum of minterms $m_1 + m_3 + m_5 + m_6$



		y			
		m_0	m_1	m_3	m_2
x		m_4	m_5	m_7	m_6
z					

Solutions for Practice K-map 1

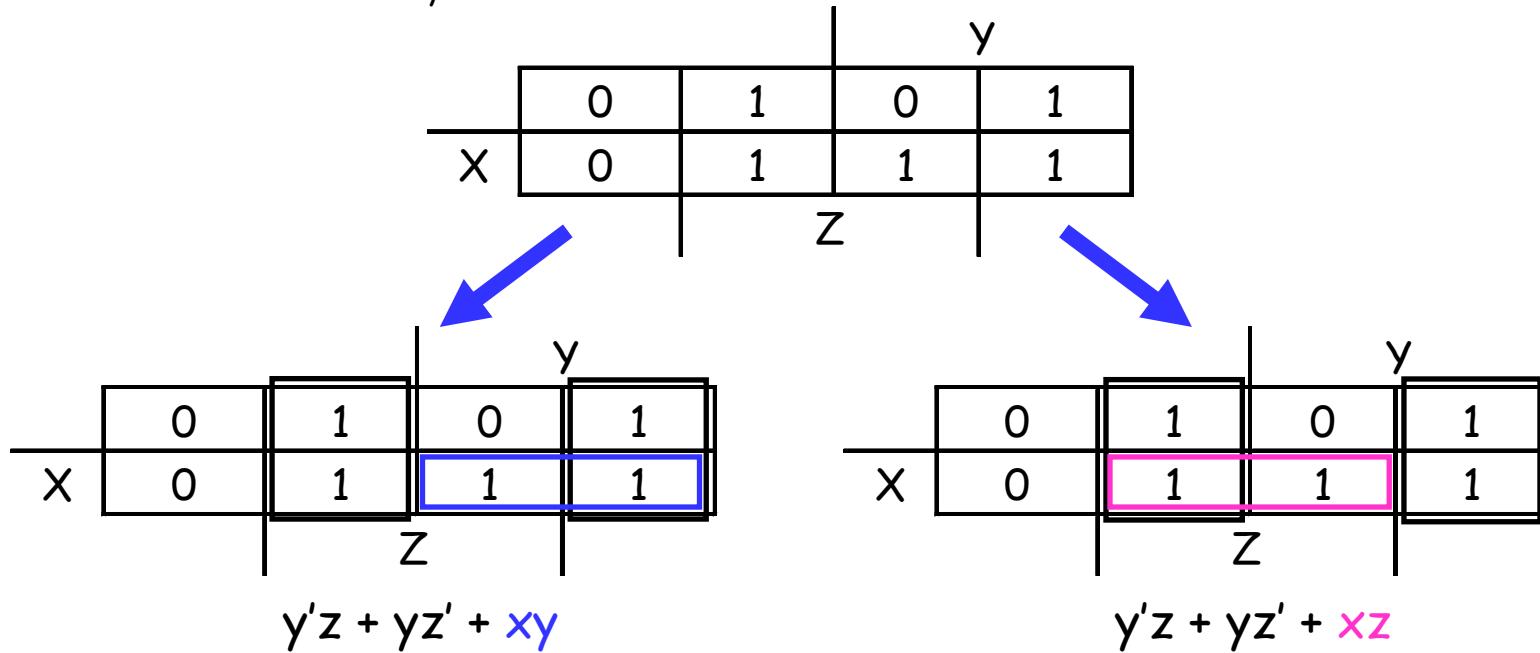
- Here is the filled in K-map, with all groups shown
 - The magenta and green groups overlap, which makes each of them as large as possible
 - Minterm m_6 is in a group all by its lonesome



- The final MSP here is $x'z + y'z + xyz'$

K-maps can be tricky!

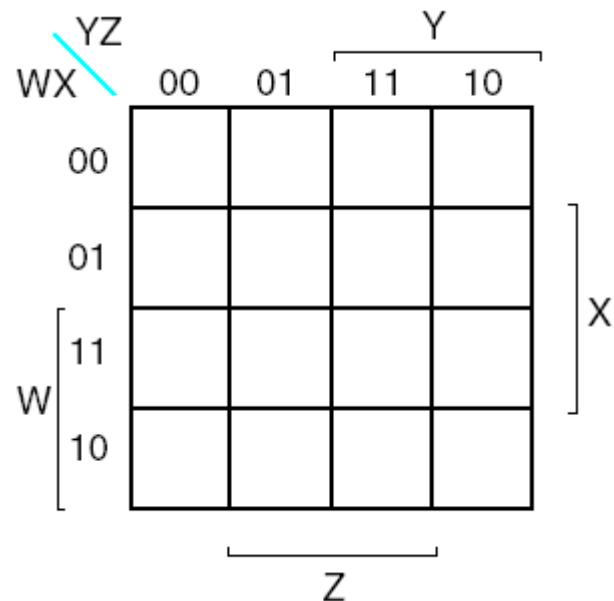
- There may not necessarily be a *unique* MSP. The K-map below yields two valid and equivalent MSPs, because there are two possible ways to include minterm m_7



- Remember that overlapping groups is possible, as shown above

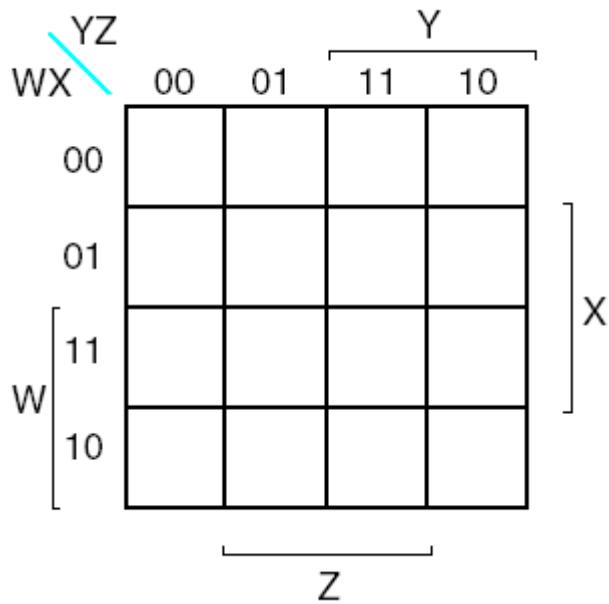
Four-variable K-maps

- We can do four-variable expressions too!
 - The minterms in the third and fourth columns, *and* in the third and fourth rows, are switched around.
 - Again, this ensures that adjacent squares have common literals



- Grouping minterms is similar to the three-variable case, but:
 - You can have rectangular groups of 1, 2, 4, 8 or 16 minterms
 - You can wrap around *all four* sides

Four-variable K-maps



		Y			
		$w'x'y'z'$	$w'x'y'z$	$w'x'yz$	$w'x'yz'$
W	X	$w'xy'z'$	$w'xy'z$	$w'xyz$	$w'xyz'$
	Z	$wxy'z'$	$wxy'z$	$wxyz$	$wxyz'$
		$wx'y'z'$	$wx'y'z$	$wx'yz$	$wx'yz'$

		Y			
		m_0	m_1	m_3	m_2
W	X	m_4	m_5	m_7	m_6
	Z	m_{12}	m_{13}	m_{15}	m_{14}
		m_8	m_9	m_{11}	m_{10}

Example: Simplify $m_0 + m_2 + m_5 + m_8 + m_{10} + m_{13}$

- The expression is already a sum of minterms, so here's the K-map:

		y			
	w	1	0	0	1
	x	0	1	0	0
		1	0	0	1
	z				

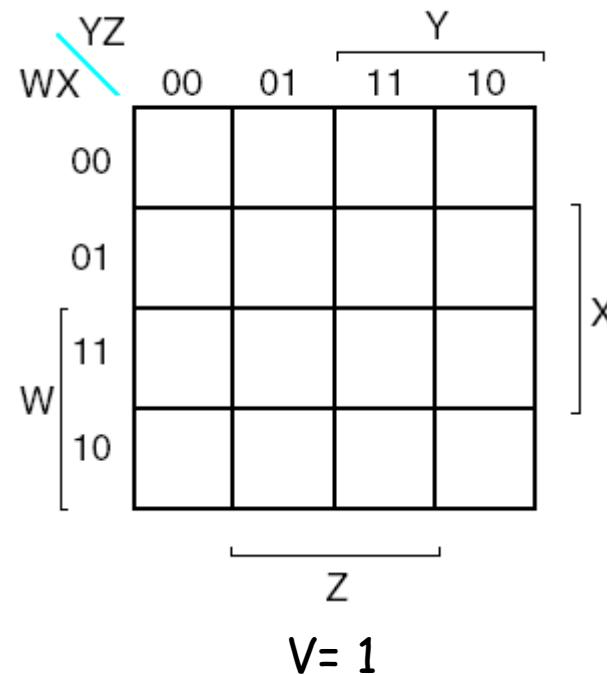
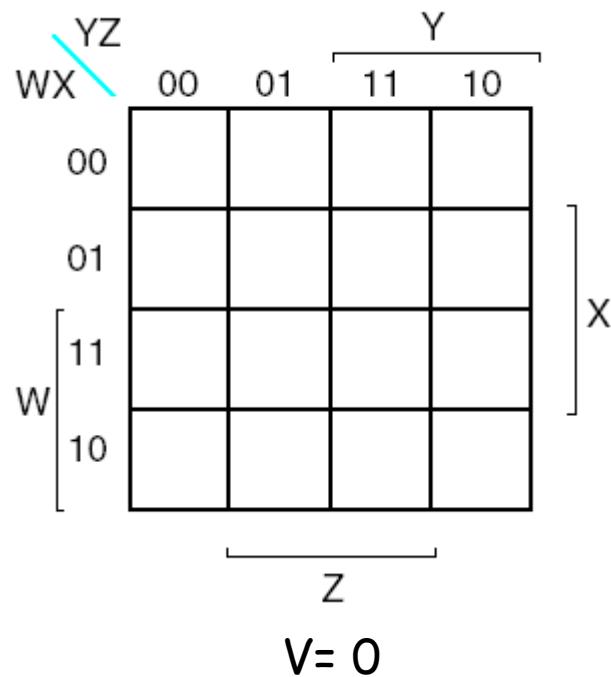
		y			
	w	m_0	m_1	m_3	m_2
	x	m_4	m_5	m_7	m_6
		m_{12}	m_{13}	m_{15}	m_{14}
	z	m_8	m_9	m_{11}	m_{10}

- We can make the following groups, resulting in the MSP $x'z' + xy'z$

		y			
	w	1	0	0	1
	x	0	1	0	0
		0	1	0	0
	z	1	0	0	1

		y			
	w	$w'x'y'z'$	$w'x'y'z$	$w'x'yz$	$w'x'yz'$
	x	$w'xy'z'$	$w'xy'z$	$w'xyz$	$w'xyz'$
		$wxy'z'$	$wxy'z$	$wxyz$	$wxyz'$
	z	$wx'y'z'$	$wx'y'z$	$wx'yz$	$wx'yz'$

Five-variable K-maps



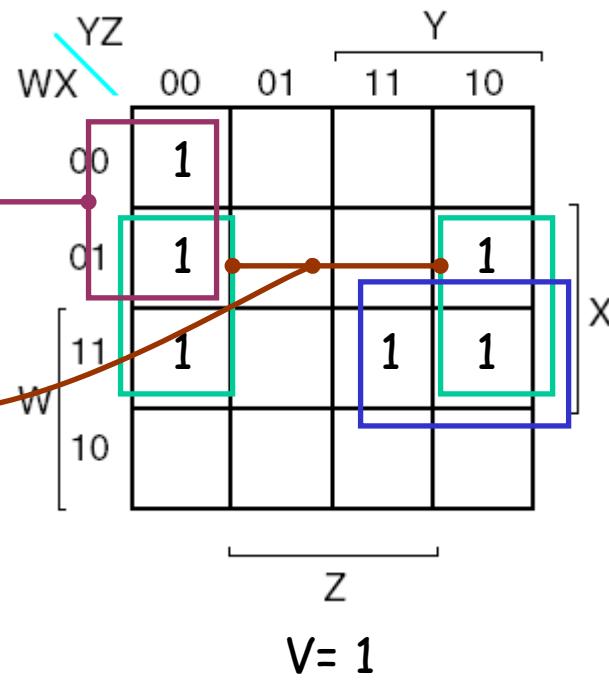
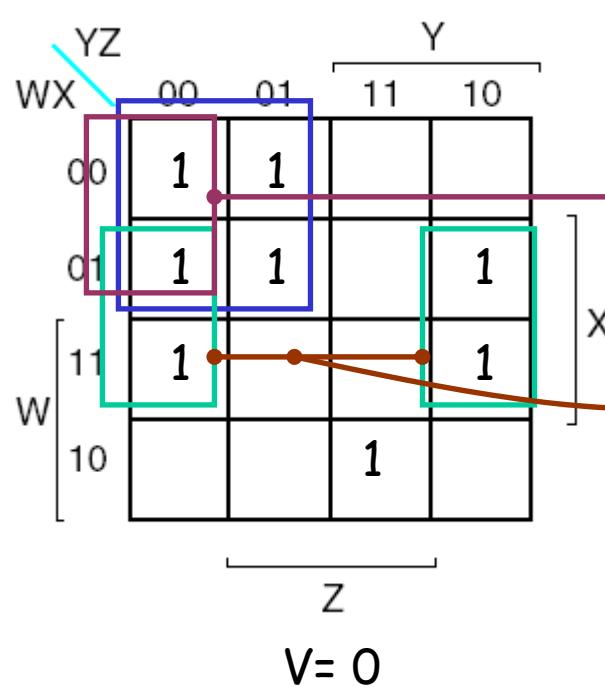
A five-variable Karnaugh map for $V=0$ showing minterms. The vertical axis is labeled W with values 00, 01, 11, and 10. The horizontal axis is labeled Z with values 00, 01, 11, and 10. The top horizontal axis is labeled y with values 00, 01, 11, and 10. The left vertical axis is labeled X with values 00, 01, 11, and 10. The minterms are labeled as follows:

m_0	m_1	m_3	m_2
m_4	m_5	m_7	m_6
m_{12}	m_{13}	m_{15}	m_{14}
m_8	m_9	m_{11}	m_{10}

A five-variable Karnaugh map for $V=1$ showing minterms. The vertical axis is labeled W with values 00, 01, 11, and 10. The horizontal axis is labeled Z with values 00, 01, 11, and 10. The top horizontal axis is labeled y with values 00, 01, 11, and 10. The left vertical axis is labeled X with values 00, 01, 11, and 10. The minterms are labeled as follows:

m_{16}	m_{17}	m_{19}	m_8
m_{20}	m_{21}	m_{23}	m_{22}
m_{28}	m_{29}	m_{31}	m_{30}
m_{24}	m_{25}	m_{27}	m_{26}

Simplify $f(V,W,X,Y,Z) = \sum m(0,1,4,5,6,11,12,14,16,20,22,28,30,31)$



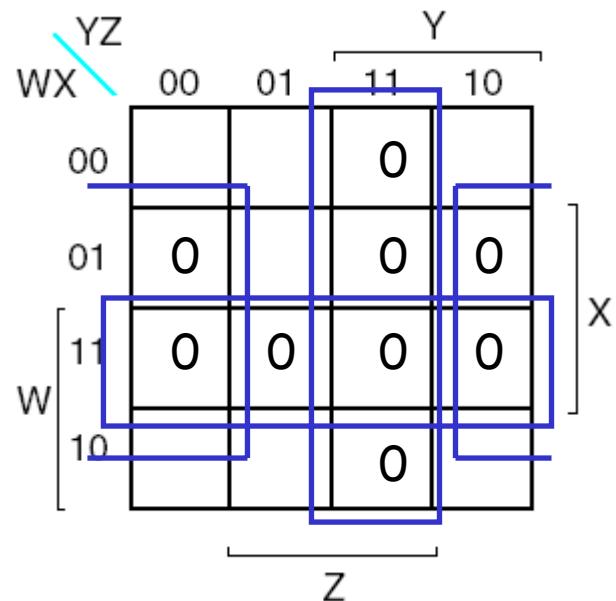
$$\begin{aligned} f = & XZ' \\ & + V'W'Y' \\ & + W'Y'Z' \\ & + VWXY \\ & + V'WX'YZ \end{aligned}$$

$$\begin{aligned} & \Sigma m(4, 6, 12, 14, 20, 22, 28, 30) \\ & \Sigma m(0, 1, 4, 5) \\ & \Sigma m(0, 4, 16, 20) \\ & \Sigma m(30, 31) \\ & m_{11} \end{aligned}$$

PoS Optimization from SoP

$$F(W, X, Y, Z) = \sum m(0, 1, 2, 5, 8, 9, 10)$$

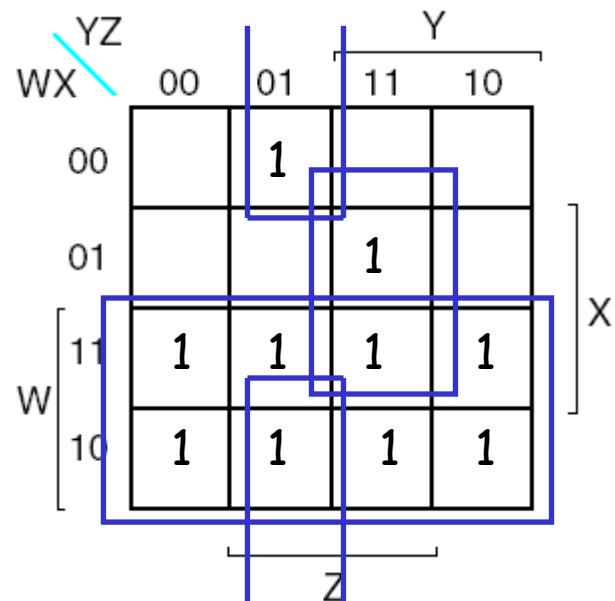
$$= \prod M(3, 4, 6, 7, 11, 12, 13, 14, 15)$$



$$F(W, X, Y, Z) = (W' + X')(Y' + Z')(X' + Z)$$

SoP Optimization from PoS

$$\begin{aligned} F(W,X,Y,Z) &= \prod M(0,2,3,4,5,6) \\ &= \sum m(1,7,8,9,10,11,12,13,14,15) \end{aligned}$$



$$F(W,X,Y,Z) = W + XYZ + X'Y'Z$$

I don't care!

- You don't always need all 2^n input combinations in an n-variable function
 - If you can guarantee that certain input combinations never occur
 - If some outputs aren't used in the rest of the circuit
- We mark don't-care outputs in truth tables and K-maps with Xs.

x	y	z	f(x,y,z)
0	0	0	0
0	0	1	1
0	1	0	X
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	X
1	1	1	1

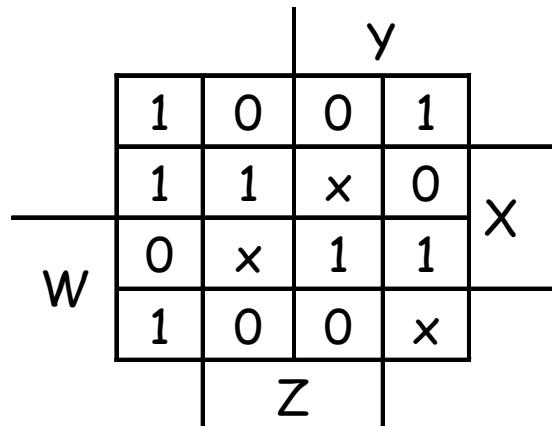
- Within a K-map, each X can be considered as either 0 or 1. You should pick the interpretation that allows for the most simplification.

Practice K-map 3

- Find a MSP for

$$f(w,x,y,z) = \sum m(0,2,4,5,8,14,15), d(w,x,y,z) = \sum m(7,10,13)$$

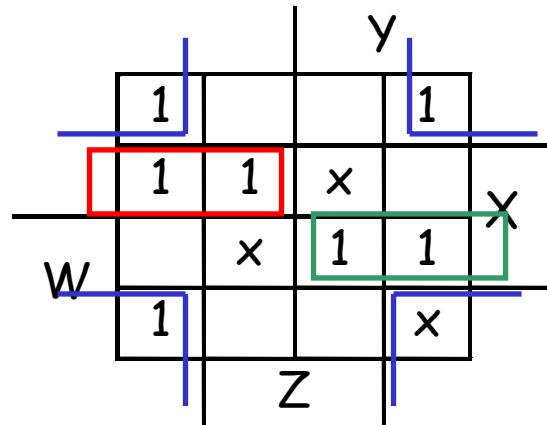
This notation means that input combinations $wxyz = 0111, 1010$ and 1101 (corresponding to minterms m_7, m_{10} and m_{13}) are unused.



Solutions for Practice K-map 3

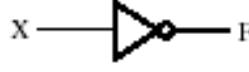
- Find a MSP for:

$$f(w,x,y,z) = \sum m(0,2,4,5,8,14,15), d(w,x,y,z) = \sum m(7,10,13)$$

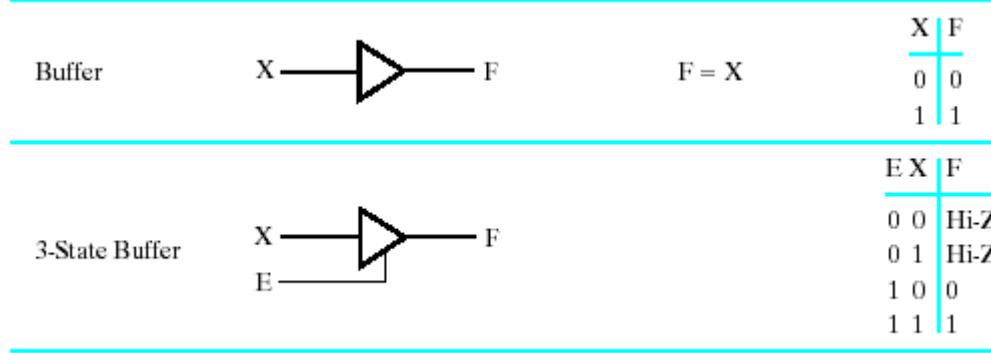


$$f(w,x,y,z) = x'z' + w'xy' + wx'y$$

AND, OR, and NOT

Graphics Symbols																		
Name	Distinctive shape	Algebraic equation	Truth table															
AND		$F = XY$	<table border="1"><thead><tr><th>X</th><th>Y</th><th>F</th></tr></thead><tbody><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></tbody></table>	X	Y	F	0	0	0	0	1	0	1	0	0	1	1	1
X	Y	F																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
OR		$F = X + Y$	<table border="1"><thead><tr><th>X</th><th>Y</th><th>F</th></tr></thead><tbody><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></tbody></table>	X	Y	F	0	0	0	0	1	1	1	0	1	1	1	1
X	Y	F																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
NOT (inverter)		$F = \overline{X}$	<table border="1"><thead><tr><th>X</th><th>F</th></tr></thead><tbody><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></tbody></table>	X	F	0	1	1	0									
X	F																	
0	1																	
1	0																	

Buffer and 3-State Buffer



- Buffer is used to amplify an electrical signal
 - Reconstructing the signal
 - More gates to be attached to the output
- Three state buffer
 - E (Enable): Controls the output
 - Hi-Z: High impedance

NAND and NOR

NAND



$$F = \overline{X \cdot Y}$$

X	Y	F
0	0	1
0	1	1
1	0	1
1	1	0

NOR



$$F = \overline{X + Y}$$

X	Y	F
0	0	1
0	1	0
1	0	0
1	1	0

- NAND: Not AND, NOR: Not OR
- Both NAND and NOR are universal gates
- Universal gate: A gate that alone can be used to implement all Boolean functions
- It is sufficient to show that NAND (NOR) can be used to implement AND, OR, and NOT operations

NANDs are special!

- The NAND gate is **universal**: it can replace all other gates!

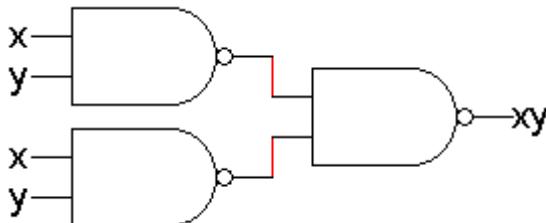
- NOT



$$(xx)' = x'$$

[because $xx = x$]

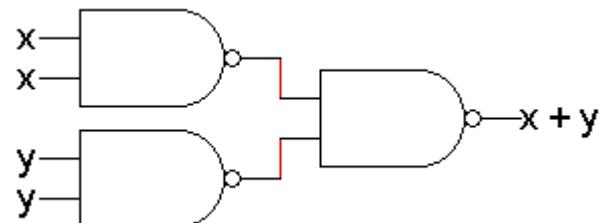
- AND



$$((xy)' (xy)')' = xy$$

[from NOT above]

- OR



$$\begin{aligned} ((xx)' (yy)')' &= (x' y')' \\ &= x + y \end{aligned}$$

[$xx = x$, and $yy = y$]
[DeMorgan's law]

XOR and XNOR

Graphics Symbols																		
Name	Distinctive shape symbol	Algebraic equation	Truth table															
Exclusive-OR (XOR)		$F = XY' + \overline{X}Y \\ = X \oplus Y$	<table border="1"> <thead> <tr> <th>X</th> <th>Y</th> <th>F</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> </tr> </tbody> </table>	X	Y	F	0	0	0	0	1	1	1	0	1	1	1	0
X	Y	F																
0	0	0																
0	1	1																
1	0	1																
1	1	0																
Exclusive-NOR (XNOR)		$F = XY + \overline{X}\overline{Y} \\ = \overline{X \oplus Y}$	<table border="1"> <thead> <tr> <th>X</th> <th>Y</th> <th>F</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>1</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> </tr> </tbody> </table>	X	Y	F	0	0	1	0	1	0	1	0	0	1	1	1
X	Y	F																
0	0	1																
0	1	0																
1	0	0																
1	1	1																

- Exclusive-OR (XOR): $X \oplus Y = XY' + X'Y$
- Exclusive-NOR (XNOR): $(X \oplus Y)' = XY + X'Y'$

$$X \oplus 0 = X$$

$$X \oplus 1 = X'$$

$$X \oplus X = 0$$

$$X \oplus X' = 1$$

$$X \oplus Y' = (X \oplus Y)' \quad X' \oplus Y = (X \oplus Y)'$$

$$X \oplus Y = Y \oplus X$$

$$(X \oplus Y) \oplus Z = X \oplus (Y \oplus Z) = X \oplus Y \oplus Z$$

More on XOR

- The general XOR function is true when an odd number of its arguments are true
- For example, we can use Boolean algebra to simplify a three-input XOR to the following expression and truth table.

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

$$= x \oplus (y'z + yz')$$

$$= x'(y'z + yz') + x(y'z + yz)'$$

$$= x'y'z + x'yz' + x(y'z + yz)'$$

$$= x'y'z + x'yz' + x((y'z)'(yz)')$$

$$= x'y'z + x'yz' + x((y + z')(y' + z))$$

$$= x'y'z + x'yz' + x(yz + y'z)$$

$$= x'y'z + x'yz' + xyz + xy'z'$$

[Definition of XOR]

[Definition of XOR]

[Distributive]

[DeMorgan's]

[DeMorgan's]

[Distributive]

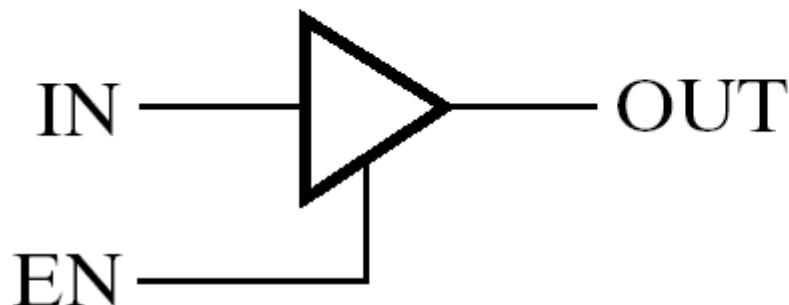
[Distributive]

x	y	z	$x \oplus y \oplus z$
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

High-Impedance Outputs

- Gates with only output values logic 0 and logic 1
 - The output is connected to either Vcc or Gnd
- A third output value: High-Impedance (Hi-Z, Z, or z)
 - The output behaves as an open-circuit, i.e., it appears to be disconnected
- Gates with Hi-Z output values can have their outputs connected together if no two gates drive the line at the same time to opposite 0 and 1 values
- Gates with only logic 0 and logic 1 outputs cannot have their outputs connected together

Three-State Buffers

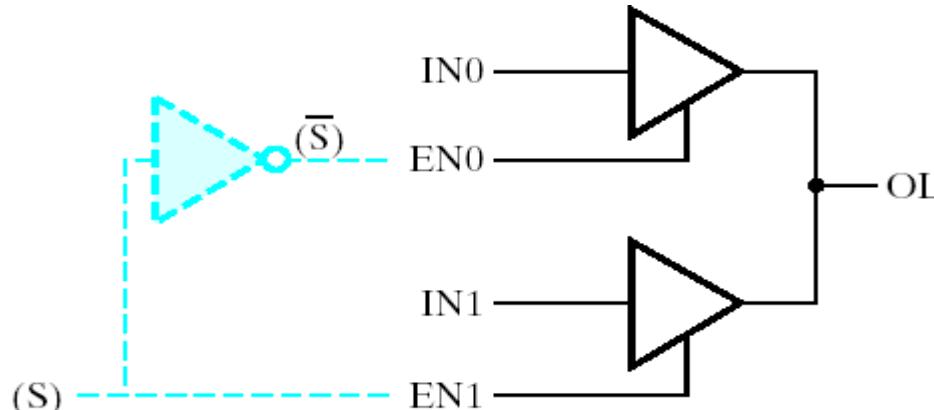


(a) Logic symbol

EN	IN	OUT
0	X	Hi-Z
1	0	0
1	1	1

(b) Truth table

Three-State Buffers

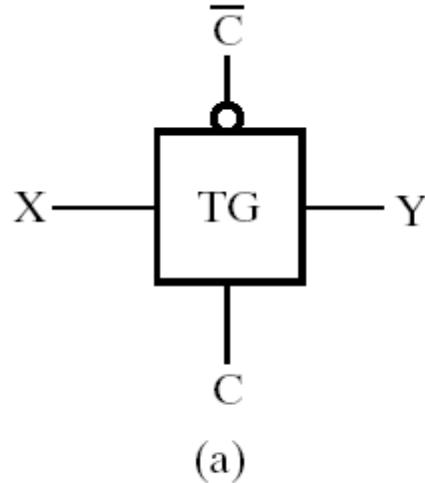


(a) Logic Diagram

EN1	EN0	IN1	IN0	OL
0	0	X	X	Hi-Z
(S)	0	(\bar{S})	1	X 0 0
0	1	X	1	1
1	0	0	X	0
1	0	1	X	1
1	1	0	0	0
1	1	1	1	1
1	1	0	1	
1	1	1	0	

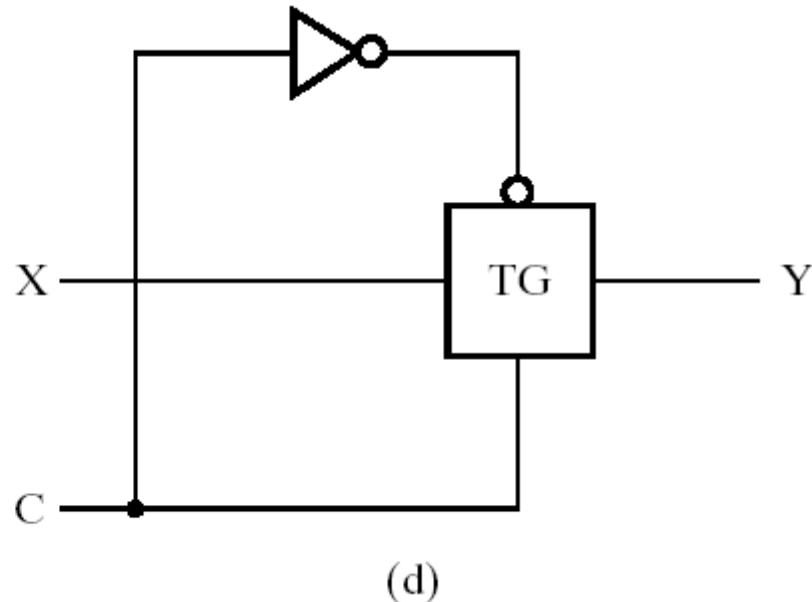
(b) Truth table

Transmission Gate



X —————— Y
C = 1 and $\bar{C} = 0$
(b)

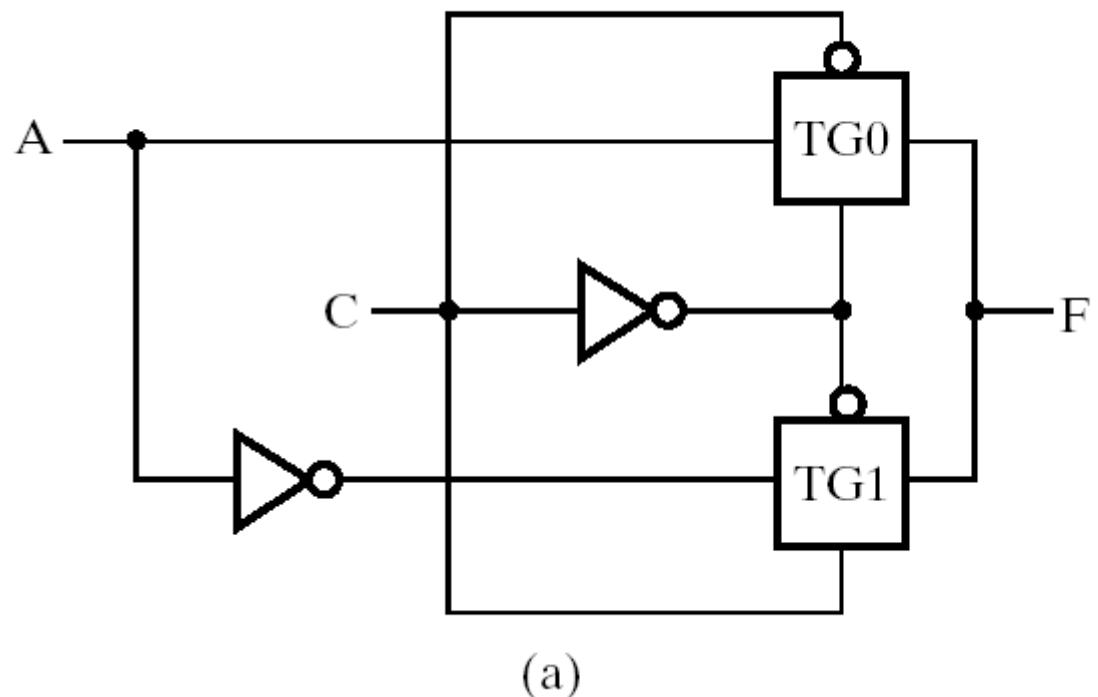
X —————— Y
C = 0 and $\bar{C} = 1$
(c)



If $C = 1$ ($C' = 0$) $\Rightarrow Y = X$

If $C = 0$ ($C' = 1$) $\Rightarrow Y = \text{Hi-Z}$

Transmission Gate XOR



A	C	TG1	TG0	F
0	0	No path	Path	0
0	1	Path	No path	1
1	0	No path	Path	1
1	1	Path	No path	0

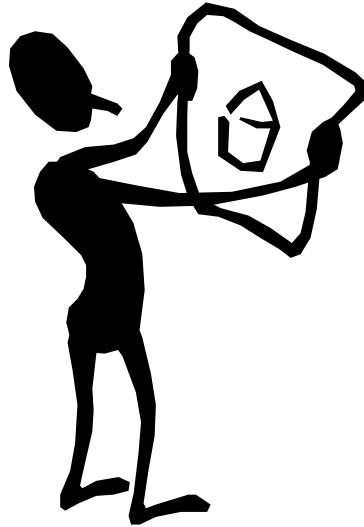
(b)

K-map Summary

- K-maps are an alternative to algebra for simplifying expressions
 - The result is a *minimal sum of products*, which leads to a minimal two-level circuit
 - It's easy to handle don't-care conditions
 - K-maps are really only good for manual simplification of small expressions...
- Things to keep in mind:
 - Remember the correct order of minterms on the K-map
 - When grouping, you can wrap around all sides of the K-map, and your groups can overlap
 - Make as few rectangles as possible, but make each of them as large as possible. This leads to fewer, but simpler, product terms
 - There may be more than one valid solution

Chapter 3: Combinational Logic Design

Introduction

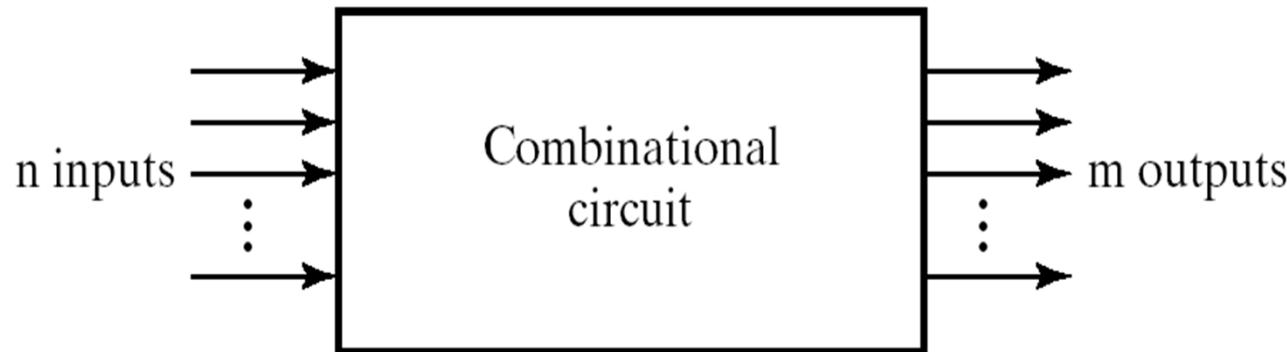


- We have learned all the prerequisite material:
 - Truth tables and Boolean expressions describe functions
 - Expressions can be converted into hardware circuits
 - Boolean algebra and K-maps help simplify expressions and circuits
- Now, let us put all of these foundations to good use, to analyze and design some larger circuits

Introduction

- Logic circuits for digital systems may be
 - Combinational
 - Sequential
- A **combinational circuit** consists of logic gates whose outputs at any time are determined by the current input values, i.e., it has no memory elements
- A **sequential circuit** consists of logic gates whose outputs at any time are determined by the current input values as well as the past input values, i.e., it has memory elements

Introduction

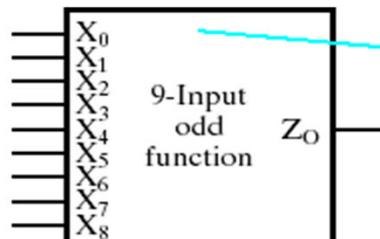


- Each input and output variable is a binary variable
- 2^n possible binary input combinations
- One possible binary value at the output for each input combination
- A truth table or m Boolean functions can be used to specify input-output relation

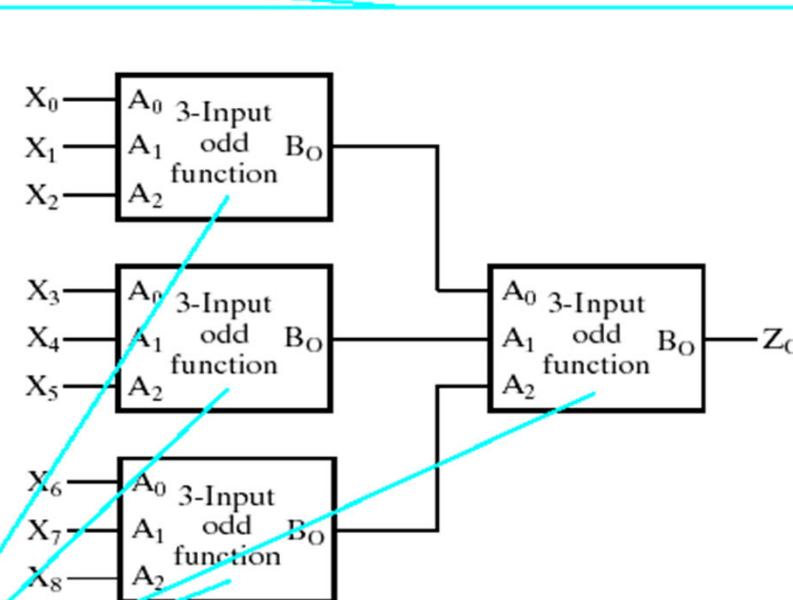
Design Hierarchy

- A single very large-scale integrated (VLSI) processos circuit contains several tens of millions of gates!
- Imagine interconnecting these gates to form the processor
- No complex circuit can be designed simply by interconnecting gates one at a time
- **Divide and Conquer** approach is used to deal with the complexity
 - Break up the circuit into pieces (*blocks*)
 - Define the functions and the interfaces of each block such that the circuit formed by interconnecting the blocks obeys the original circuit specification
 - If a block is still too large and complex to be designed as a single entity, it can be broken into smaller blocks

Divide and Conquer



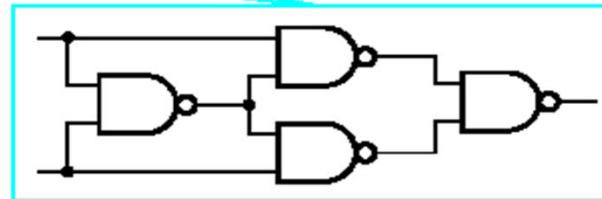
(a) Symbol for circuit



(b) Circuit as interconnected 3-input odd function blocks

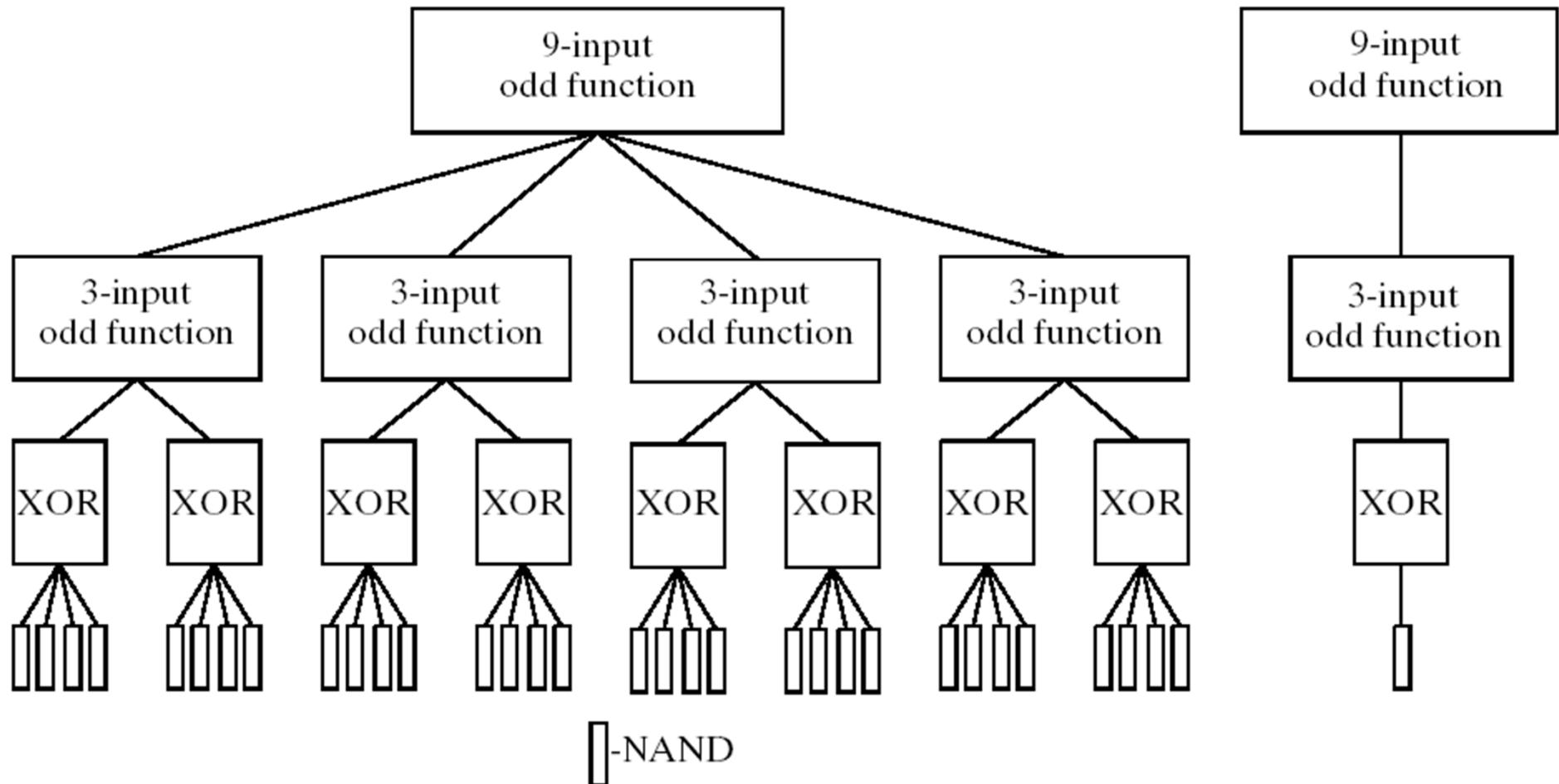


(c) 3-input odd function circuit as interconnected exclusive-OR blocks



(d) Exclusive-OR block as interconnected NANDs

Hierarchical Design due to Divide and Conquer



Hierarchical Design

- A hierarchy reduce the complexity required to represent the schematic diagram of a circuit
- In any hierarchy, the leaves consist of predefined blocks, some of which may be primitives. No need to design a predefined block!
 - A *primitive block* is the one with a logic symbol, but no logic schematic
 - Primitive blocks such as gates are of *predefined blocks*
 - More complex structures can also be defined as *predefined blocks*
- The blocks can be reused; for a block reused, only one design is required
 - Instance: Appearance of a design within a block
 - Instantiation: Using a block in the design

Designing Complex Circuits

- Computer-Aided Design (CAD) tools
 - Schematic capture tools: Support the drawing of blocks and interconnections at all levels of the hierarchy
 - Libraries of graphic symbols
 - Logic Simulator
- Hardware Description Languages (HDLs)
 - VHDL and Verilog, both are the IEEE standard
 - VHDL: Very High Speed Integrated Circuits (VHSIC) HDL
 - Like programming languages, but tuned to describe hardware structures and behavior
 - Alternative to schematics (structural description)
 - Behavioral description also possible
 - Logic synthesis: RTL of a system -> Netlist (structural description)

Levels of Integration

- Digital circuits are constructed with integrated circuits
- An **integrated circuit (IC)** is a silicon semiconductor crystal (informally a chip) containing the electronic components for the digital gates and storage elements
- **Small-scale integrated (SSI)**: Primitive gates, # of gates < 10
- **Medium-scale integrated (MSI)**: Elementary digital functions (4-bit addition), $10 < \# \text{ of gates} < 100$
- **Large-scale integrated (LSI)**: Small processors, small memories, programmable modules, $100 < \# \text{ of gates} < \text{a few thousand}$
- **Very large-scale integrated (VLSI)**: Complex microprocessors and digital signal processing chips, several thousand to tens of millions of gates

Design Procedure

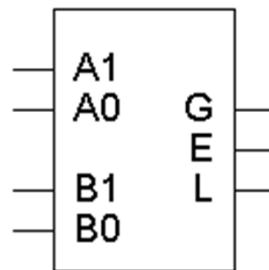
- The design of a combinational circuit involves the following steps:
 - Specification: How the circuit operates is clearly expressed
 - Formulation: Derivation of the truth table or the Boolean equations that define the relationship between inputs and outputs
 - Optimization: Algebraic or K-map optimization of the truth table and draw the corresponding logic diagram
 - Technology Mapping: Transform the logic diagram to a new diagram using the available implementation technology
 - Verification: Verify the correctness of the final design

Comparing 2-bit Numbers - Specification

- Let's design a circuit that compares two 2-bit numbers, A and B. The circuit should have three outputs:
 - G ("Greater") should be 1 only when $A > B$
 - E ("Equal") should be 1 only when $A = B$
 - L ("Lesser") should be 1 only when $A < B$
- Make sure you understand the problem
 - Inputs A and B will be 00, 01, 10, or 11 (0, 1, 2 or 3 in decimal)
 - For any inputs A and B, exactly one of the three outputs will be 1

Comparing 2-bit Numbers - Specification

- Two 2-bit numbers means a total of four inputs
 - We should name each of them
 - Let's say the first number consists of digits A1 and A0 from left to right, and the second number is B1 and B0
- The problem specifies three outputs: G, E and L



Comparing 2-bit Numbers - Formulation

- For this problem, it's probably easiest to start with a truth table. This way, we can explicitly show the relationship ($>$, $=$, $<$) between inputs
- A four-input function has a sixteen-row truth table
- It's usually clearest to put the truth table rows in binary numeric order; in this case, from 0000 to 1111 for A1, A0, B1 and B0
- Example: $01 < 10$, so the sixth row of the truth table (corresponding to inputs A=01 and B=10) shows that output L=1, while G and E are both 0.

A1	A0	B1	B0	G	E	L
0	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	0	1
0	1	1	1			
1	0	0	0			
1	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			

Comparing 2-bit Numbers - Formulation

A1	A0	B1	B0	G	E	L
0	0	0	0	0	1	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	0	1
0	1	0	0	1	0	0
0	1	0	1	0	1	0
0	1	1	0	0	0	1
0	1	1	1	0	0	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	0	1	1	0	0	1
1	1	0	0	1	0	0
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	0	1	0

Comparing 2-bit Numbers - Optimization

- Let's use K-maps. There are *three* functions (each with the same inputs $A_1 A_0 B_1 B_0$), so we need *three* K-maps

		B1		
		0	0	
		0	0	
		1	0	
		1	0	
		1	1	
		1	1	
				B0

A_1

A_0

		B1		
		0	0	
		0	0	
		1	0	
		0	1	
		0	0	
		0	0	
				B0

A_1

A_0

		B1		
		0	0	
		0	0	
		1	1	
		0	0	
		0	0	
		0	0	
				B0

A_1

A_0

$$G(A_1, A_0, B_1, B_0) = \\ A_1 A_0 B_0' + \\ A_0 B_1' B_0' + \\ A_1 B_1'$$

$$E(A_1, A_0, B_1, B_0) = \\ A_1' A_0' B_1' B_0' + \\ A_1' A_0 B_1' B_0 + \\ A_1 A_0 B_1 B_0 + \\ A_1 A_0' B_1 B_0'$$

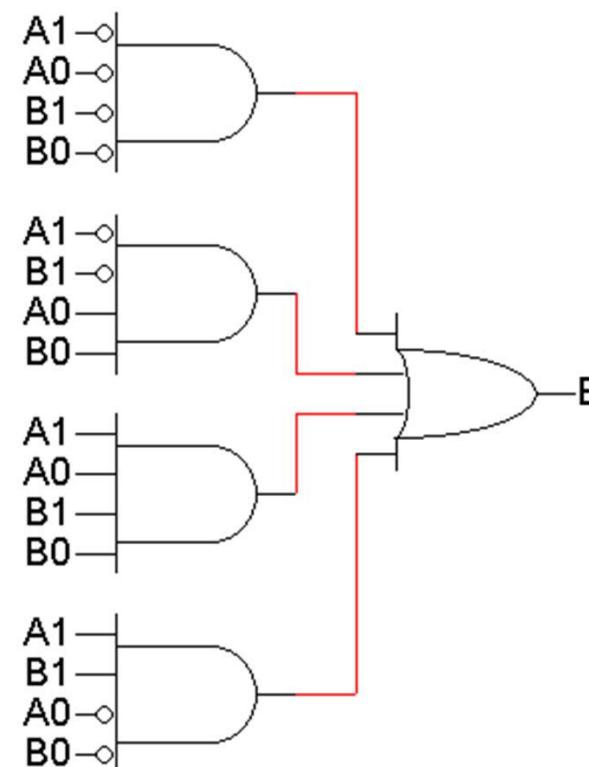
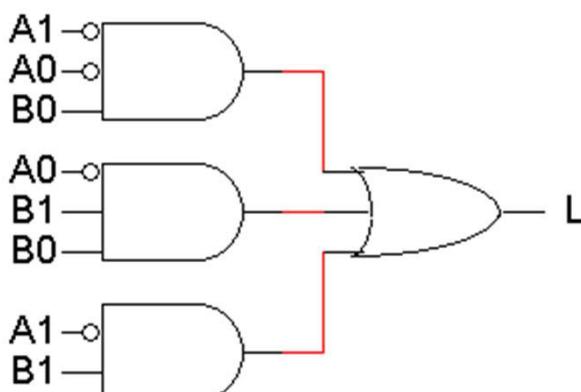
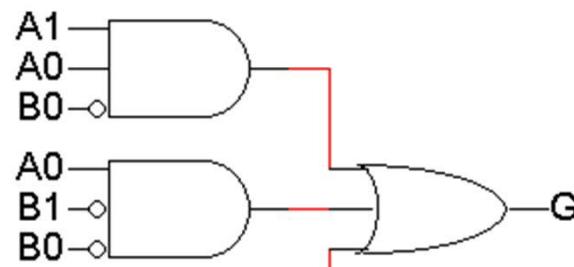
$$L(A_1, A_0, B_1, B_0) = \\ A_1' A_0' B_0 + \\ A_0' B_1 B_0 + \\ A_1' B_1$$

Comparing 2-bit Numbers - Optimization

$$G = A_1 A_0 B_0' + A_0 B_1' B_0' + A_1 B_1'$$

$$E = A_1' A_0' B_1' B_0' + A_1' A_0 B_1' B_0 + A_1 A_0 B_1 B_0 + A_1 A_0' B_1 B_0'$$

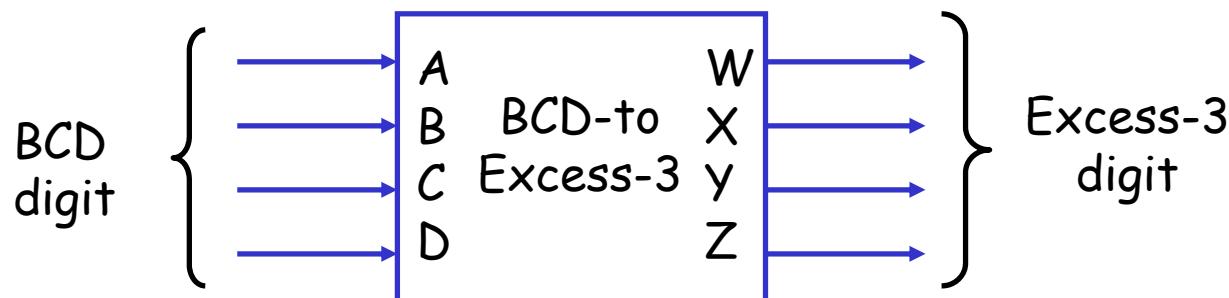
$$L = A_1' A_0' B_0 + A_0' B_1 B_0 + A_1' B_1$$



BCD-to-Excess-3 Code Converter - Specification

The excess-3 code for a decimal digit is the binary combination corresponding to the decimal digit plus 3. For example, the excess-3 code for decimal digit 5 is the binary combination for $5 + 3 = 8$, which is 1000.

Each BCD digit is four bits with the bits, from most significant to least significant, labeled A, B, C, D. Each excess-3 digit is four bits, with the bits, from most significant to least significant, labeled W, X, Y, Z.

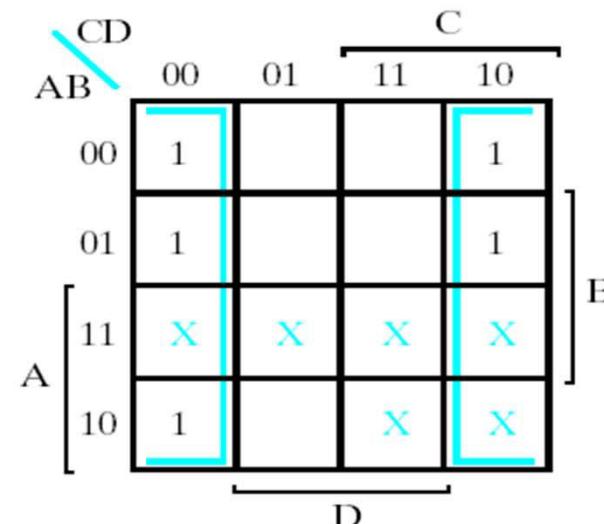
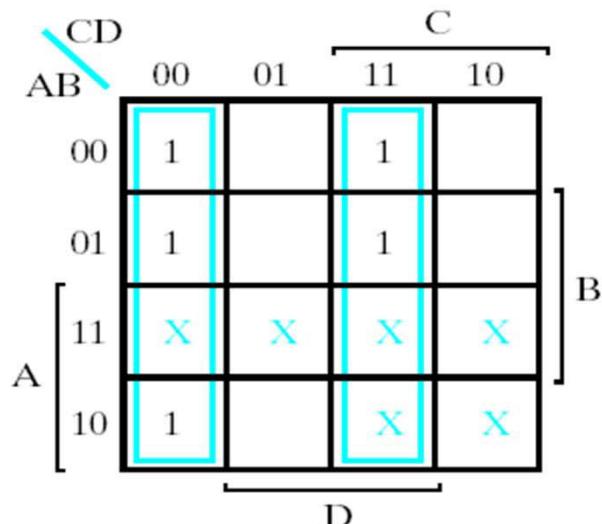
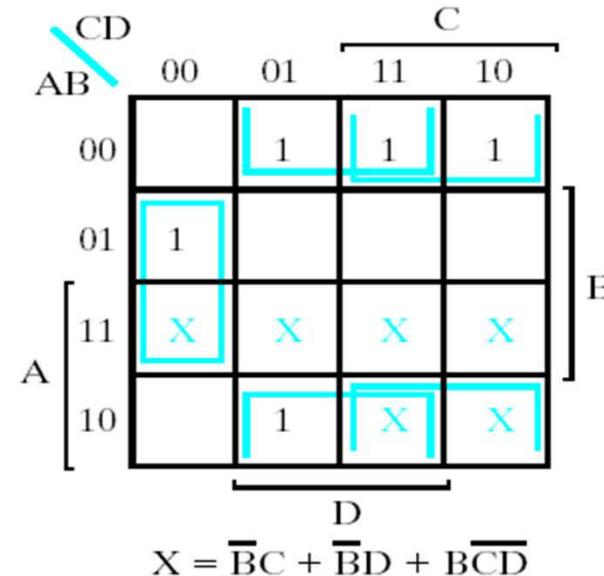
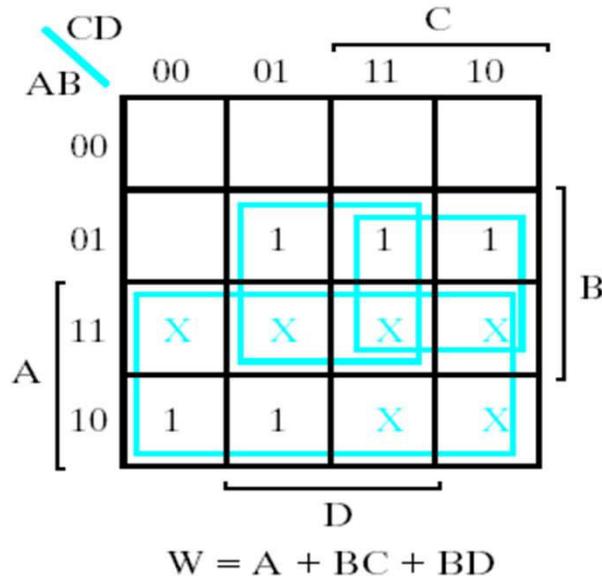


BCD-to-Excess-3 Code Converter - Formulation

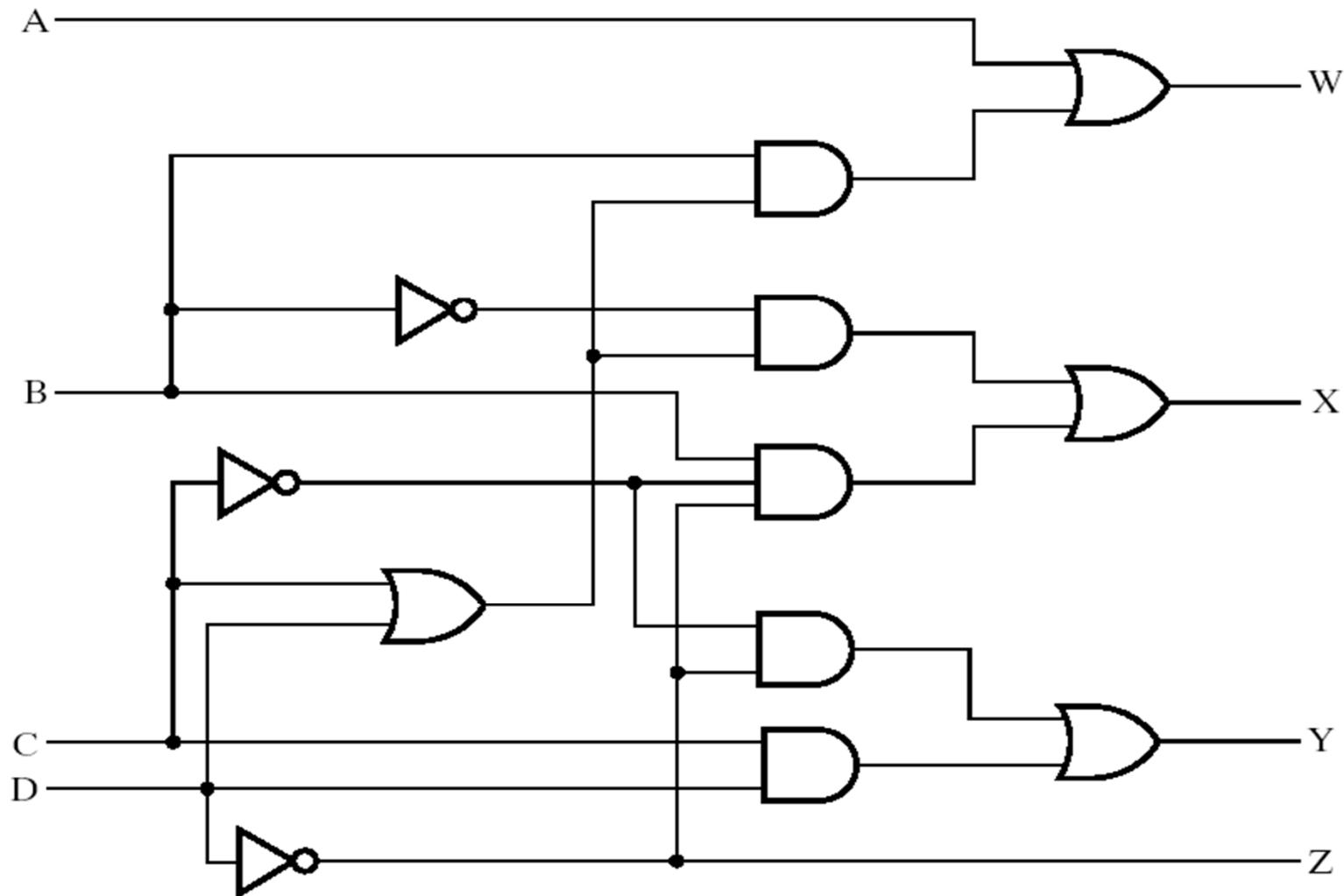
□ TABLE 3-1
Truth Table for Code Converter Example

Decimal Digit	Input BCD				Output Excess-3			
	A	B	C	D	W	X	Y	Z
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0

BCD-to-Excess-3 Code Converter - Optimization

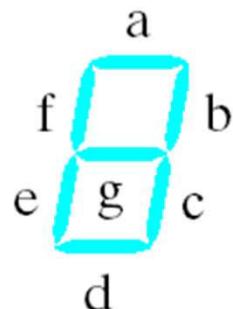


BCD-to-Excess-3 Code Converter - Optimization



BCD-to-Seven-Segment Decoder - Specification

Digital readouts found in many consumer electronic products often use Light Emitting Diodes (LEDs). Each digit of the readout is formed from seven LED segments. Each segment can be illuminated by a digital signal. A BCD-to-seven-segment decoder is a combinational circuit that accepts a decimal digit in BCD and generates the appropriate outputs for the segments of the display for the decimal digit. The seven outputs of the decoder (a,b,c,d,e,f,g) select the corresponding segments in the display. BCD-to-seven-segment decoder has four inputs, A, B, C, and D for the BCD digit and seven outputs, a through g, for controlling the segments.



(a) Segment designation



(b) Numeric designation for display

BCD-to-Seven-Segment Decoder - Formulation

□ TABLE 3-2
Truth Table for BCD- to- Seven-Segment
Decoder

BCD Input				Seven-Segment Decoder						
A	B	C	D	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
All other inputs				0	0	0	0	0	0	0

BCD-to-Seven-Segment Decoder - Optimization

$$a = A'C + A'BD + B'C'D' + AB'C'$$

$$b = A'B' + A'C'D' + A'CD + AB'C'$$

$$c = A'B + A'D + B'C'D' + AB'C'$$

$$d = A'C^D' + A'B'C + B'C'D' + AB'C' + A'BC'D$$

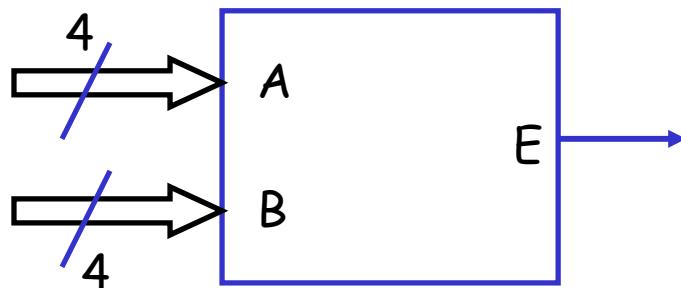
$$e = A'CD' + B'C'D'$$

$$f = A'BC' + A'C'D' + A'BD' + AB'C'$$

$$g = A'CD' + A'B'C + A'BC' + AB'C'$$

4-Bit Equality Comparator - Specification

The inputs to the circuit consist of two vectors: $A(3:0)$ and $B(3:0)$. Vector A consists of four bits, $A(3)$, $A(2)$, $A(1)$, and $A(0)$, with $A(3)$ as the msb. Vector B consists of four bits, $B(3)$, $B(2)$, $B(1)$, and $B(0)$, with $B(3)$ as the msb. The output of the circuit is a single bit variable E . Output E is equal to 1 if A and B are equal and equal to 0 if A and B are unequal.

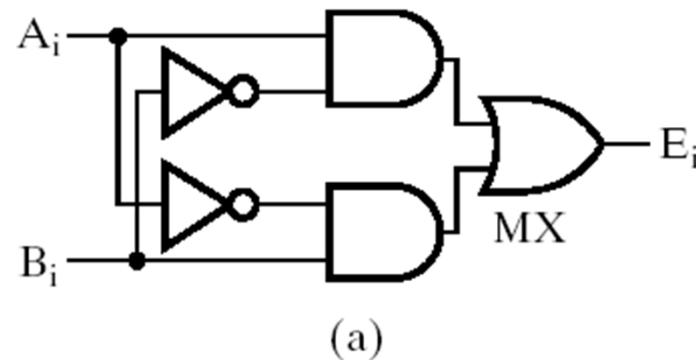


4-Bit Equality Comparator - Formulation

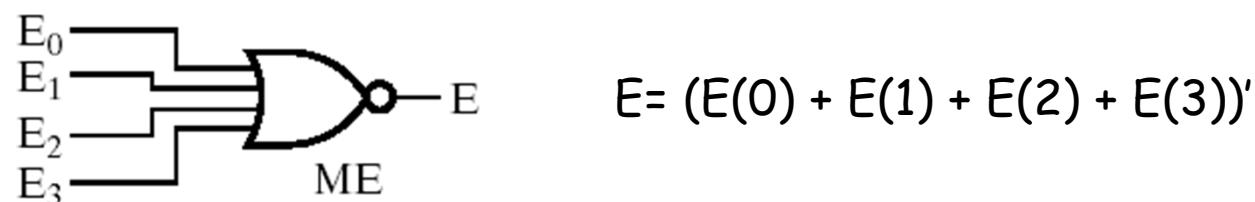
- Since there are 8 inputs, using a truth table is impractical!!!
- Apply divide and conquer design approach
- Observation: In order for A and B to be equal, the bit values in each of the respective positions, 3 down to 0, must be equal
- We need four 1-bit comparator
- We need an additional circuit to combine the outputs of 1-bit comparators

4-Bit Equality Comparator - Optimization

$A(i)$	$B(i)$	$E(i)$
0	0	0
0	1	1
1	0	1
1	1	0

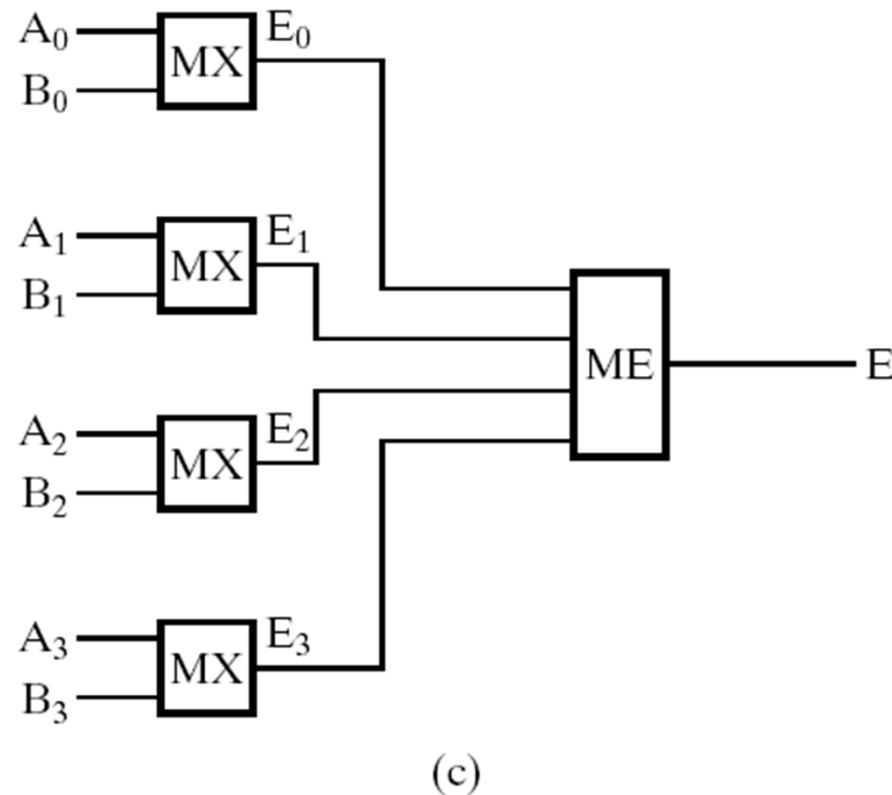


1-bit Comparator



(b)

4-Bit Equality Comparator - Optimization

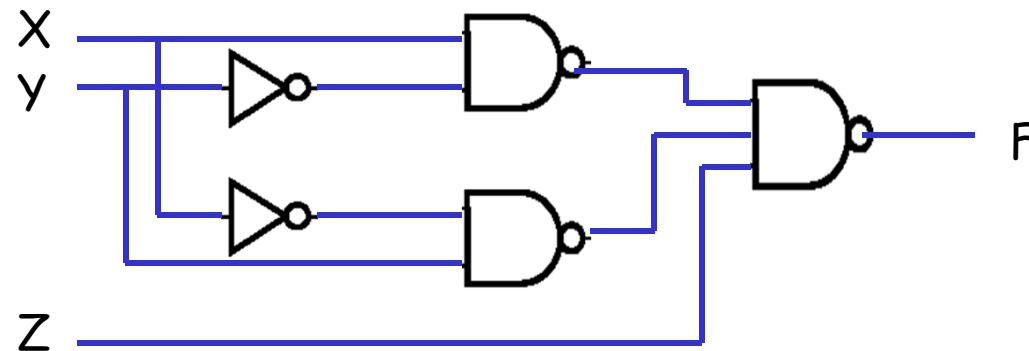


Techology Mapping - NAND Gate Implementation

- The circuit is defined in Sum-of-Products form
- Goal: Use only NAND gates to implement the circuit

$$F = XY' + X'Y + Z$$

$$\begin{aligned} F &= (F')' = [(XY' + X'Y + Z)']' \\ &= [(XY')' \cdot (X'Y)' \cdot (Z')']' \end{aligned}$$

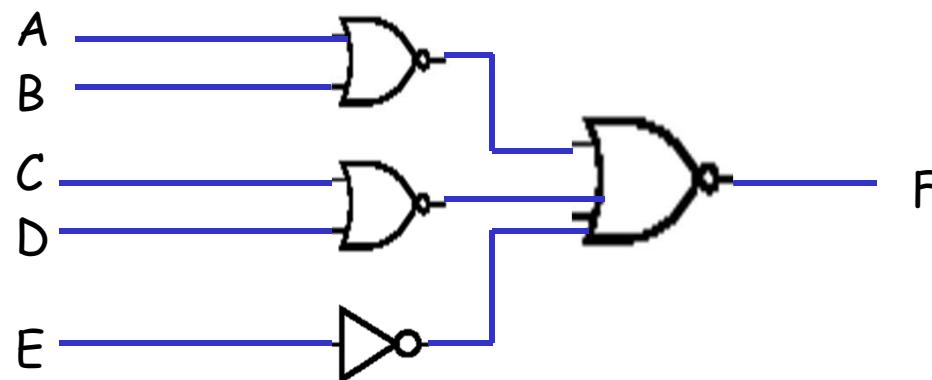


Techology Mapping - NOR Gate Implementation

- The circuit is defined in Product-of-Sums form
- Goal: Use only NOR gates to implement the circuit

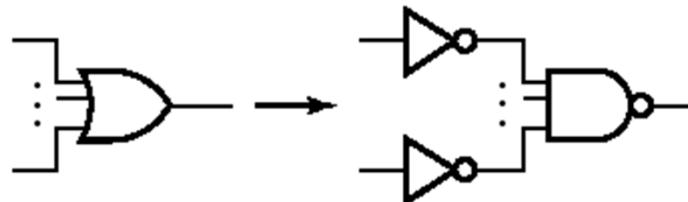
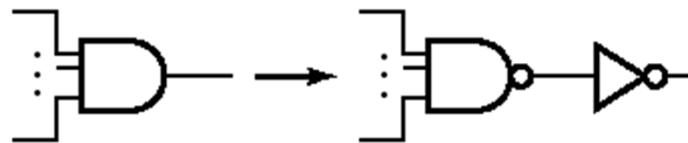
$$F = (A + B)(C + D)E$$

$$\begin{aligned} F &= (F')' = [\{ (A + B)(C + D)E \}']' \\ &= [(A + B)' + (C + D)' + E']' \end{aligned}$$

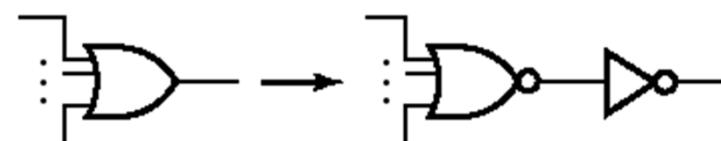
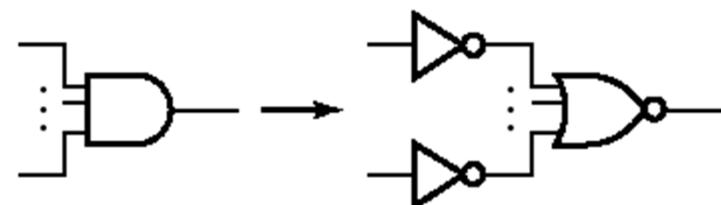


Multilevel NAND (NOR) Implementation

1. Replace each AND and OR gate with the NAND (NOR) gate and inverter equivalent circuits shown below



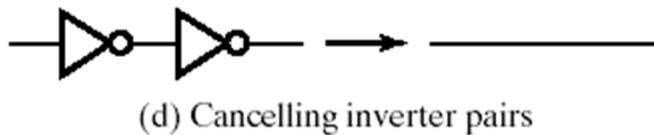
(a) Mapping to NAND gates



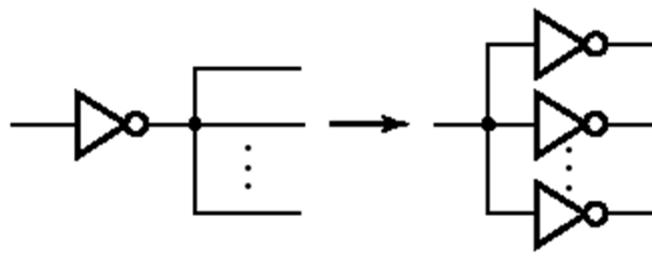
(b) Mapping to NOR gates

Multilevel NAND (NOR) Implementation

2. Cancel all inverter pairs

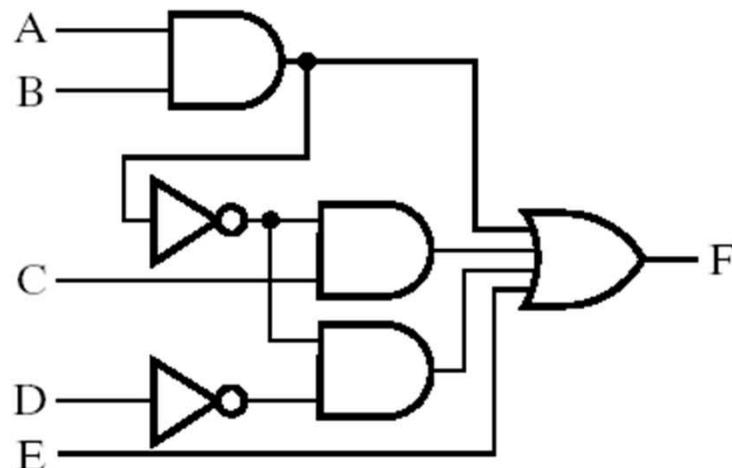


3. Without changing the logic function, (a) Push all inverters (b) Replace inverters in parallel with a single inverter that drives all of the outputs of the parallel inverters (c) Repeat a and b until there is at most one inverter between the circuit input or driving NAND(NOR) gate output and the attached NAND(NOR) gate inputs

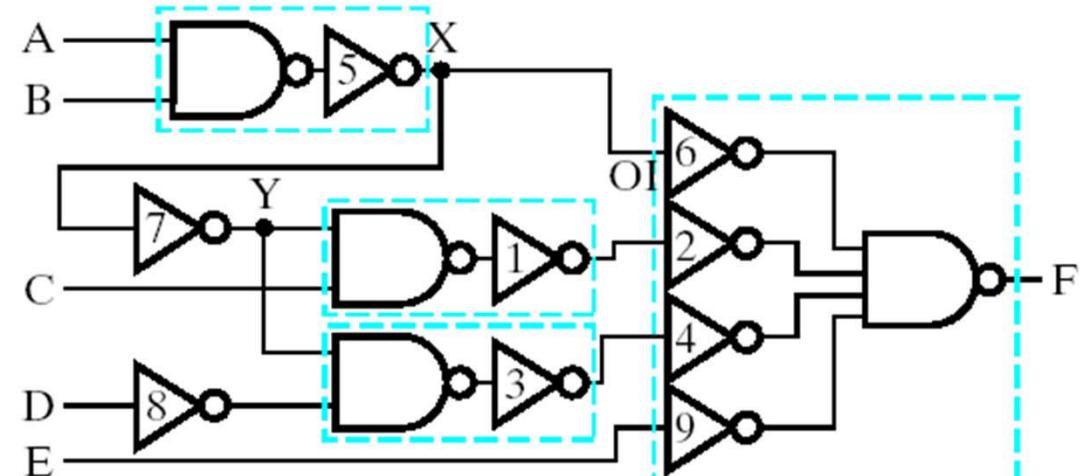


(c) Pushing an inverter through a “dot”

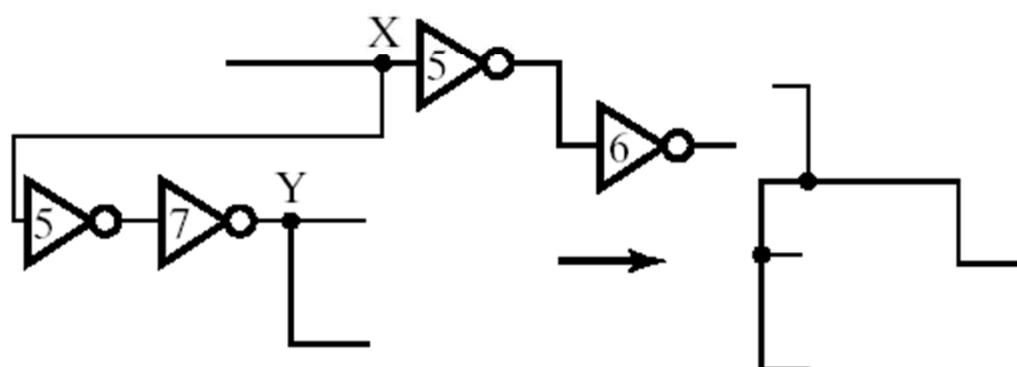
$$\text{Example} - F = AB + (AB)'C + (AB)'D' + E$$



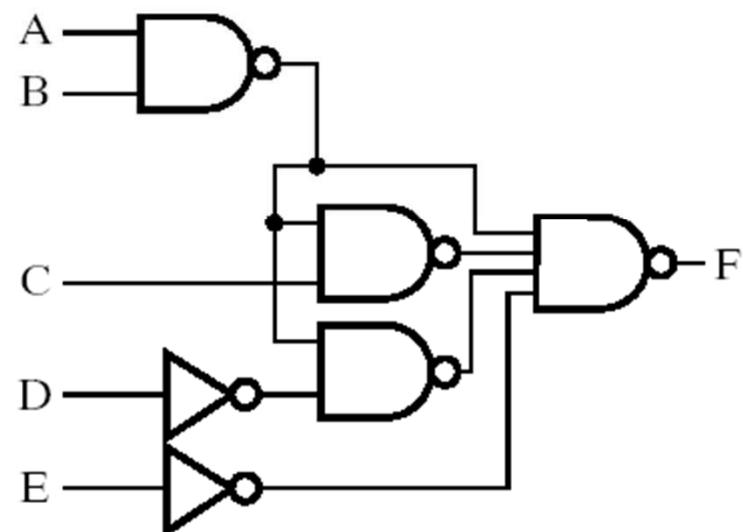
(a)



(b)

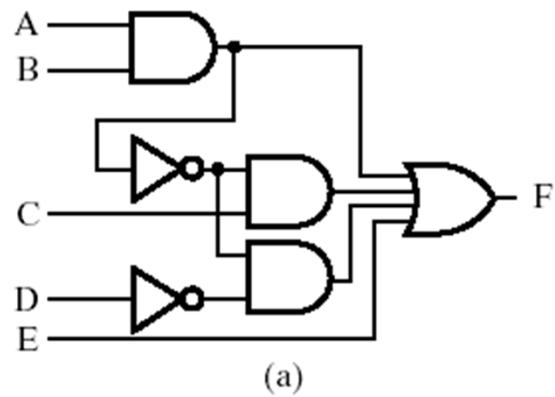


(c)

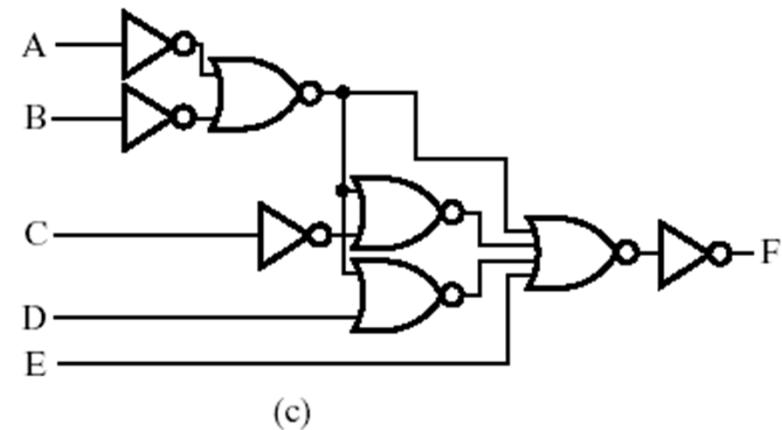


(d)

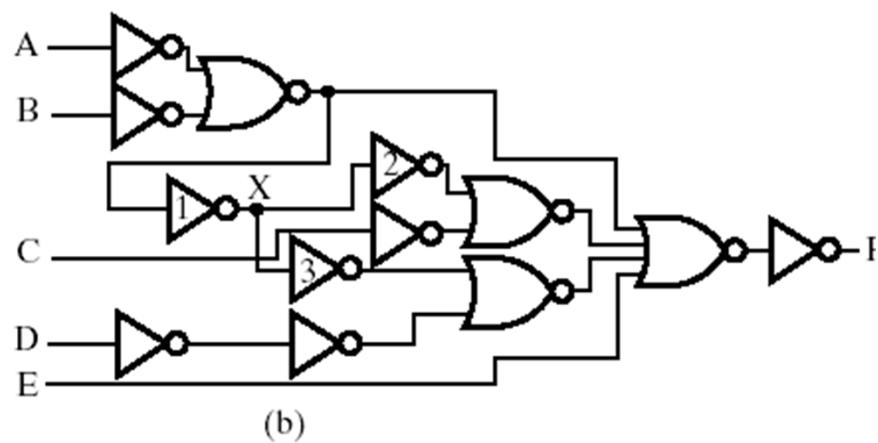
$$\text{Example} - F = AB + (AB)'C + (AB)'D' + E$$



(a)



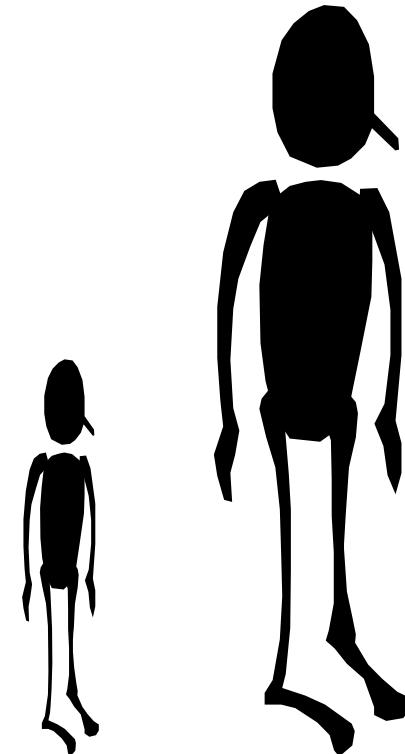
(c)



(b)

Summary

- Functions can be represented with expressions, truth tables or circuits. These are all equivalent, and we can arbitrarily transform between them
- Designing a circuit requires you to first find a (simplified) Boolean expression for the function you want to compute. You can then convert the expression into a circuit



Summary

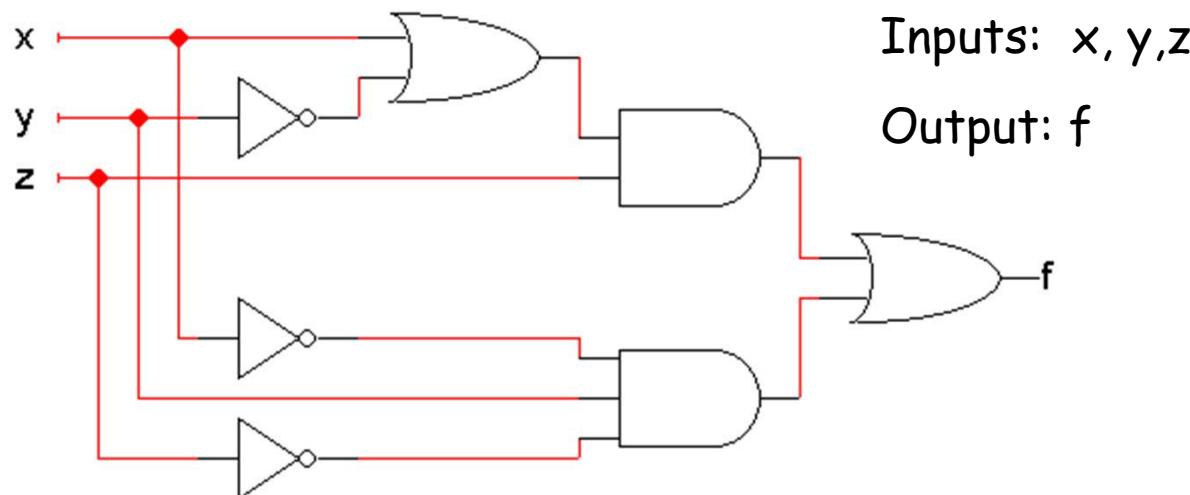
- NAND and NOR are universal gates which can replace all others
 - There are two representations for NAND gates (AND-NOT and NOT-OR), which are equivalent by DeMorgan's law
 - Similarly, there are two representations for NOR gates too
- You can convert a circuit with primitive gates into a NAND or NOR diagram by judicious use of the axiom $(x')' = x$, to ensure that you don't change the overall function

Summary

- Circuits made up of gates, that don't have any feedback, are called *combinatorial circuits*
 - No feedback: outputs are not connected to inputs
 - If you change the inputs, *and wait for a while*, the correct outputs show up
 - Why? Capacitive loading ("fill up the water level" analogy)
- So, when such ckts are used in a computer, the time it takes to get stable outputs is important
- For the same reason, a single output cannot drive too many inputs
 - Will be too slow to "fill them up"
 - May not have enough power
- So, the design criteria are:
 - Propagation delay (how many gets in a sequence from in to out)
 - Fan-out
 - Fan-in (Number of inputs to a single gate)

Verification - Circuit Analysis

- **Circuit analysis** involves figuring out what some circuit does
 - Every circuit computes some function, which can be described with Boolean expressions or truth tables
 - So, the goal is to find an expression or truth table for the circuit
- The first thing to do is to figure out what the inputs and outputs of the overall circuit are

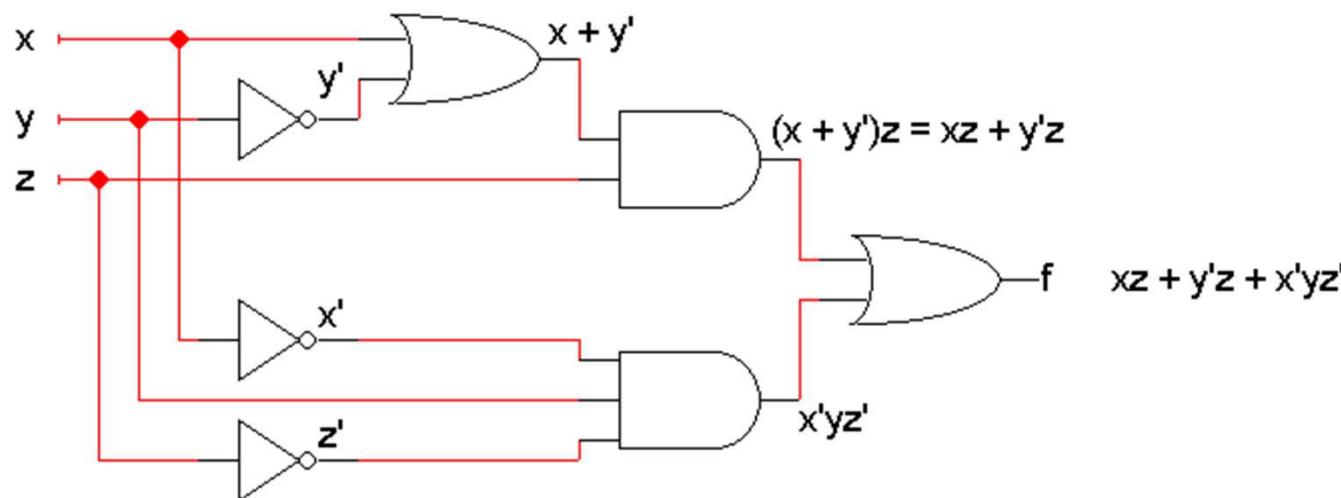


Inputs: x, y, z

Output: f

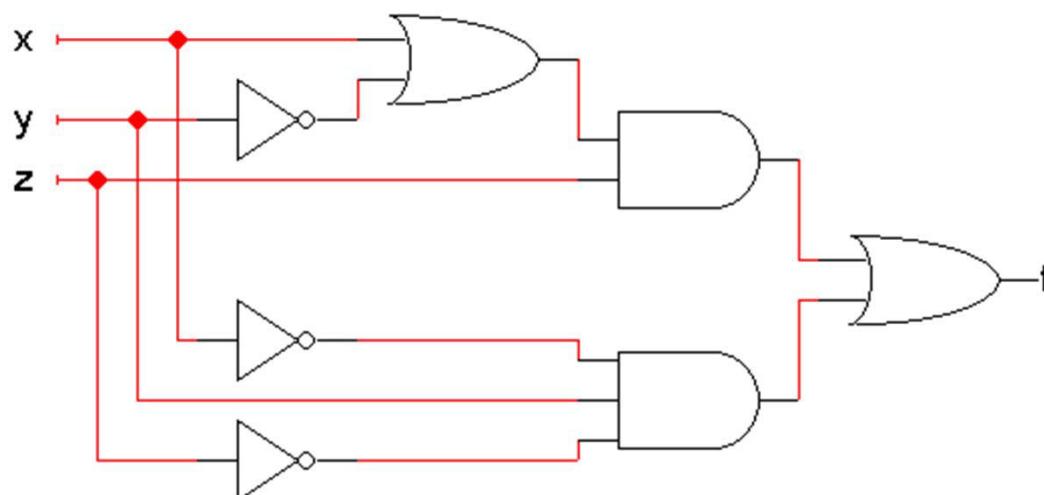
Verification - Circuit Analysis

- Write expressions for the outputs of each individual gate, based on that gate's inputs
 - Start from the inputs and work towards the outputs
 - It might help to do some algebraic simplification along the way



Verification - Circuit Analysis

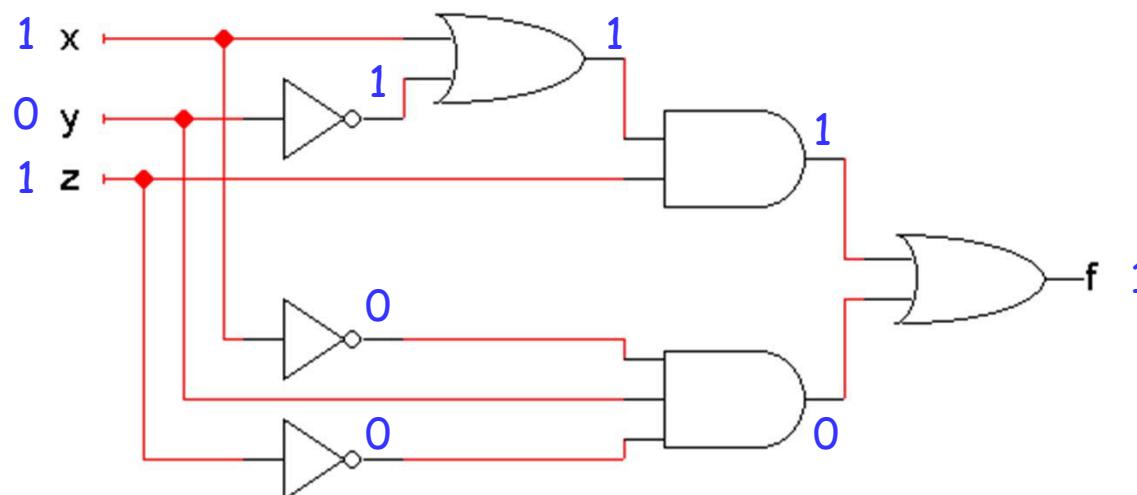
- It's also possible to find a truth table directly from the circuit.
- Once you know the number of inputs and outputs, list all the possible input combinations in your truth table
 - A circuit with n inputs should have a truth table with 2^n rows



x	y	z	f
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Verification - Circuit Analysis

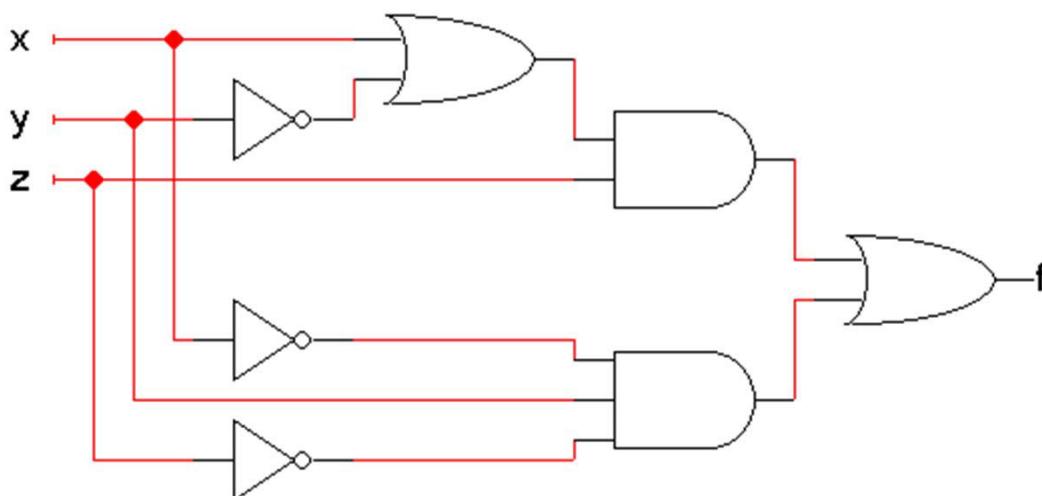
- You can simulate the circuit by hand to find the output for each possible combination of inputs



x	y	z	f
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	0
1	1	1	1

Verification - Circuit Analysis

- Doing the same thing for all the other input combinations yields the complete truth table
- This is simple, but tedious



x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Verification - Circuit Analysis

- Remember that if you already have a Boolean expression, you can use that to easily make a truth table
- For example, since we already found that the circuit computes the function $f(x,y,z) = xz + y'z + x'yz'$, we can use that to fill in a table:

x	y	z	xz	y'z	x'yz'	f
0	0	0	0	0	0	0
0	0	1	0	1	0	1
0	1	0	0	0	1	1
0	1	1	0	0	0	0
1	0	0	0	0	0	0
1	0	1	1	1	0	1
1	1	0	0	0	0	0
1	1	1	1	0	0	1

Verification - Circuit Analysis

- The opposite is also true: it's easy to come up with an expression if you already have a truth table
- Convert a truth table into a sum of minterms expression

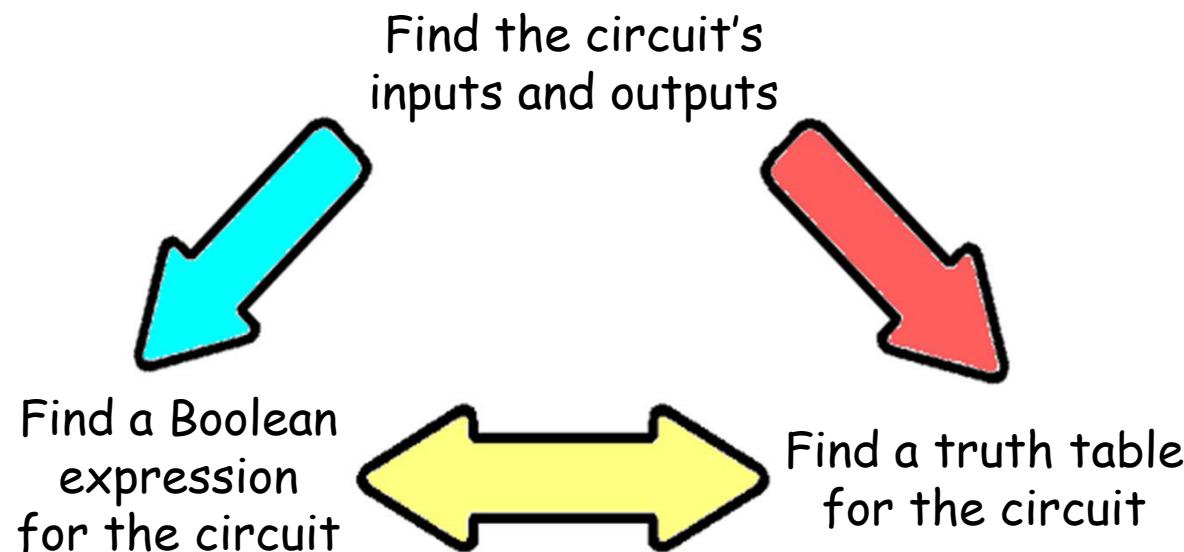
x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

$$\begin{aligned}f(x,y,z) &= x'y'z + x'yz' + xy'z + xyz \\&= m_1 + m_2 + m_5 + m_7\end{aligned}$$

- You can then simplify this sum of minterms if desired—using a K-map, for example

Circuit Analysis Summary

- After finding the circuit inputs and outputs, you can come up with either an expression or a truth table to describe what the circuit does
- You can easily convert between expressions and truth tables



Chapter 3 Ctd: Combinational Functions and Circuits

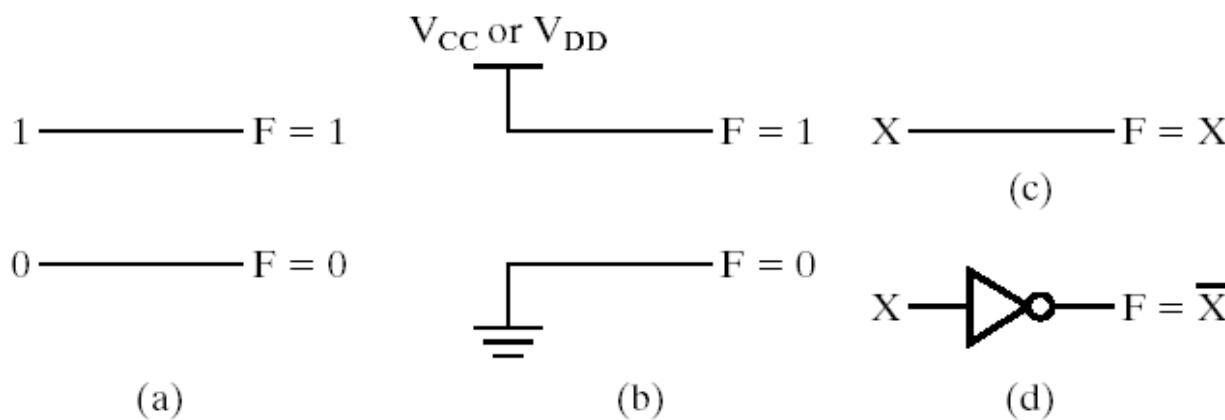
Value Fixing, Transferring, and Inverting

- Four different functions are possible as a function of single Boolean variable

□ TABLE 4-1
Functions of One Variable

X	F = 0	F = X	F = \bar{X}	F = 1
0	0	0	1	1
1	0	1	0	1

Transferring Inverting Value Fixing

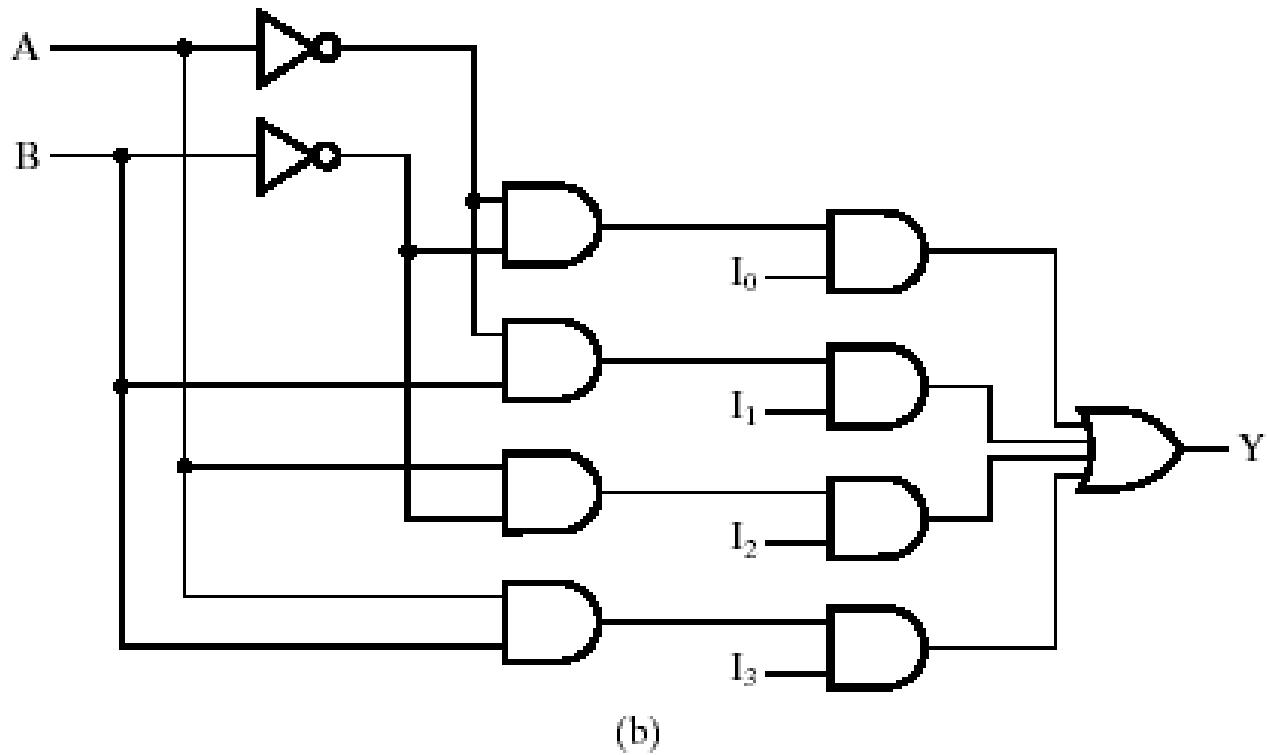


Value Fixing for Implementing a Function

- Y is actually a function of six variables - truth table with 64 rows and 7 columns
- Putting $I_0 - I_3$ in the output column reduces the truth table

A	B	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

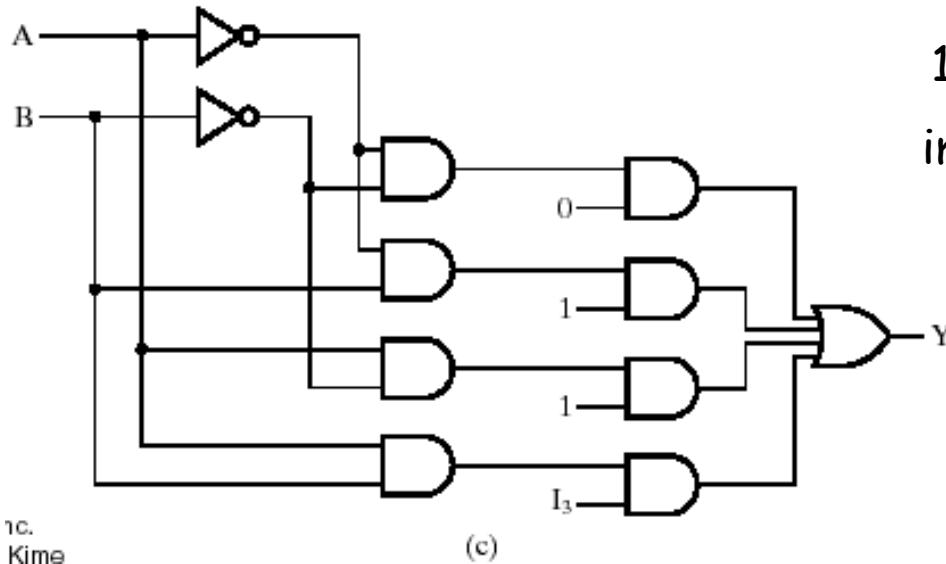
(a)



(b)

$$Y(A, B, I_0, I_1, I_2, I_3) = A'B'I_0 + A'BI_1 + AB'I_2 + ABI_3$$

Value Fixing for Implementing a Function



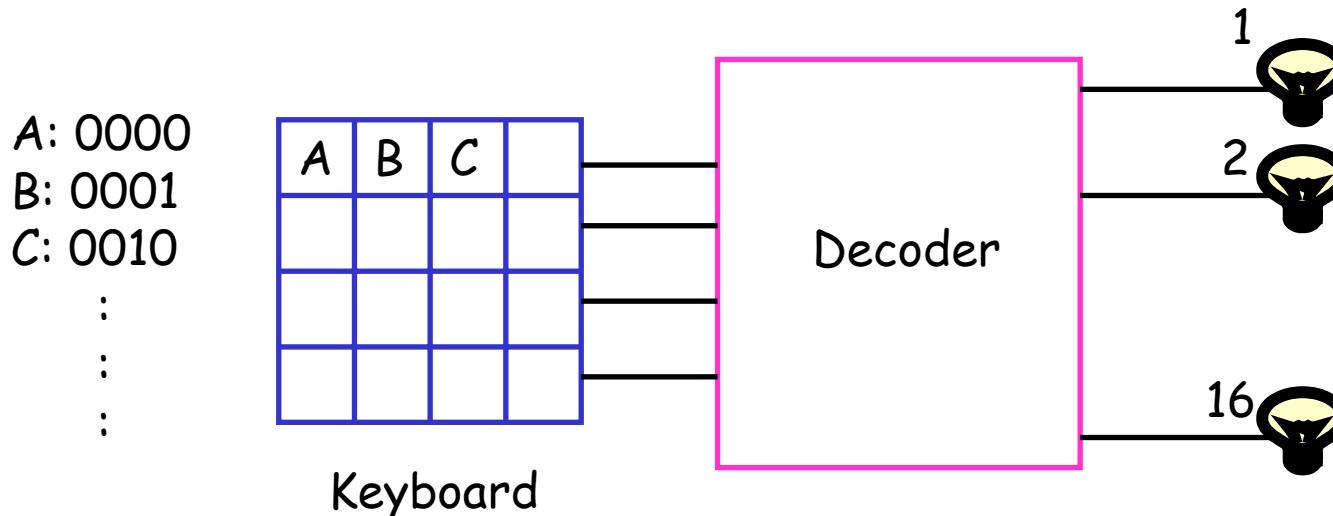
16 different functions can be implemented by setting $I_0 - I_3$ appropriately

□ TABLE 4-2
Function Implementation by Value-Fixing

A	B	$Y = A + B$	$Y = AB + \bar{A}B$	$Y = A + B$ ($I_3 = 1$) or $Y = A\bar{B} + \bar{A}B$ ($I_3 = 0$)
0	0	0	0	0
0	1	1	1	1
1	0	1	1	1
1	1	1	0	I_3

0	0	0	0	0
0	1	1	1	1
1	0	1	1	1
1	1	1	0	I_3

What is a decoder?

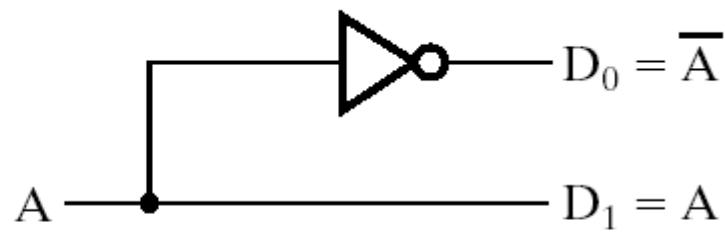


- Decoder: A combinational circuit with an n -bit binary code applied to its inputs and m -bit binary code appearing at the outputs
 - $n \leq m \leq 2^n$
 - Detect which of the 2^n combinations is represented at the inputs
 - Produce m outputs, only one of which is "1"

1-to-2 (Line) Decoder

A	D ₀	D ₁
0	1	0
1	0	1

(a)



(b)

2-to-4 Decoder

- A 2-to-4 decoder operates according to the following truth table
 - The 2-bit input is called $S_1 S_0$, and the four outputs are Q_0-Q_3
 - If the input is the binary number i , then output Q_i is uniquely true

S_1	S_0	Q_0	Q_1	Q_2	Q_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

- For instance, if the input $S_1 S_0 = 10$ (decimal 2), then output Q_2 is true, and Q_0, Q_1, Q_3 are all false
- This circuit “decodes” a binary number into a “one-of-four” code

How can you build a 2-to-4 decoder?

- Follow the design procedures from last time! We have a truth table, so we can write equations for each of the four outputs (Q0-Q3)

S1	S0	Q0	Q1	Q2	Q3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

- In this case there's not much to be simplified. Here are the equations:

$$Q_0 = S_1' S_0'$$

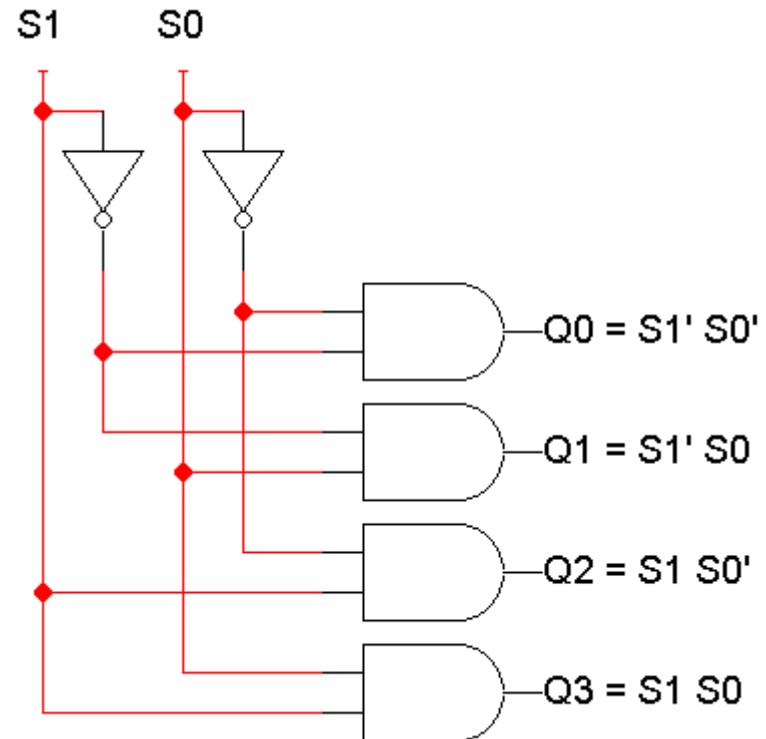
$$Q_1 = S_1' S_0$$

$$Q_2 = S_1 S_0'$$

$$Q_3 = S_1 S_0$$

2-to-4 Decoder

S1	S0	Q0	Q1	Q2	Q3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1



Enable Inputs

- Many devices have an additional **enable input**, which is used to "activate" or "deactivate" the device
- For a decoder,
 - EN=1 activates the decoder, so it behaves as specified earlier. Exactly one of the outputs will be 1
 - EN=0 "deactivates" the decoder. By convention, that means *all* of the decoder's outputs are 0
- We can include this additional input in the decoder's truth table:

EN	S1	S0	Q0	Q1	Q2	Q3
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

An aside: abbreviated truth tables

- In this table, note that whenever EN=0, the outputs are always 0, *regardless* of inputs S1 and S0.

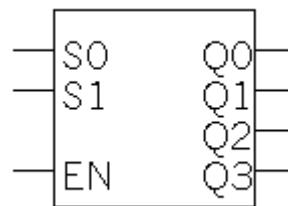
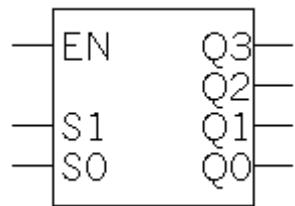
EN	S1	S0	Q0	Q1	Q2	Q3
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

- We can abbreviate the table by writing x's in the input columns for S1 and S0

EN	S1	S0	Q0	Q1	Q2	Q3
0	x	x	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

Blocks and Abstraction

- Decoders are common enough that we want to encapsulate them and treat them as an individual entity
- **Block diagrams** for 2-to-4 decoders are shown here. The *names* of the inputs and outputs, not their order, is what matters.

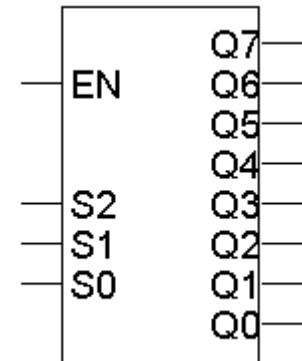


$$\begin{aligned}Q0 &= S1' S0' \\Q1 &= S1' S0 \\Q2 &= S1 S0' \\Q3 &= S1 S0\end{aligned}$$

- A decoder block provides **abstraction**:
 - You can use the decoder as long as you know its truth table or equations, without knowing exactly what's inside
 - It makes diagrams simpler by hiding the internal circuitry
 - It simplifies hardware reuse. You don't have to keep rebuilding the decoder from scratch every time you need it
- These blocks are like functions in programming!

3-to-8 decoder

- Larger decoders are similar. Here is a 3-to-8 decoder
 - The block symbol is on the right
 - A truth table (without EN) is below
 - Output equations are at the bottom right
- Again, only one output is true for any input combination



S2	S1	S0	Q0	Q1	Q2	Q3	Q4	Q5	Q6	Q7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

$$\begin{aligned}Q_0 &= S_2' S_1' S_0' \\Q_1 &= S_2' S_1' S_0 \\Q_2 &= S_2' S_1 S_0' \\Q_3 &= S_2' S_1 S_0 \\Q_4 &= S_2 S_1' S_0' \\Q_5 &= S_2 S_1' S_0 \\Q_6 &= S_2 S_1 S_0' \\Q_7 &= S_2 S_1 S_0\end{aligned}$$

So what good is a decoder?

- Do the truth table and equations look familiar?

S1	S0	Q0	Q1	Q2	Q3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

$$\begin{aligned} Q_0 &= S_1' S_0' \\ Q_1 &= S_1' S_0 \\ Q_2 &= S_1 S_0' \\ Q_3 &= S_1 S_0 \end{aligned}$$

- Decoders are sometimes called **minterm generators**
 - For each of the input combinations, exactly one output is true
 - Each output equation contains all of the input variables
 - These properties hold for all sizes of decoders
- Arbitrary functions can be implemented with decoders. If a sum of minterms equation for a function is given, a decoder (a minterm generator) is used to implement that function

Design example: Addition

- Let's make a circuit that adds three 1-bit inputs X, Y and Z
- We will need two bits to represent the total; let's call them C and S, for "carry" and "sum." Note that C and S are two separate functions of the same inputs X, Y and Z
- Here are a truth table and sum-of-minterms equations for C and S

$$0 + 1 + 1 = 10 \longrightarrow$$

X	Y	Z	C	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

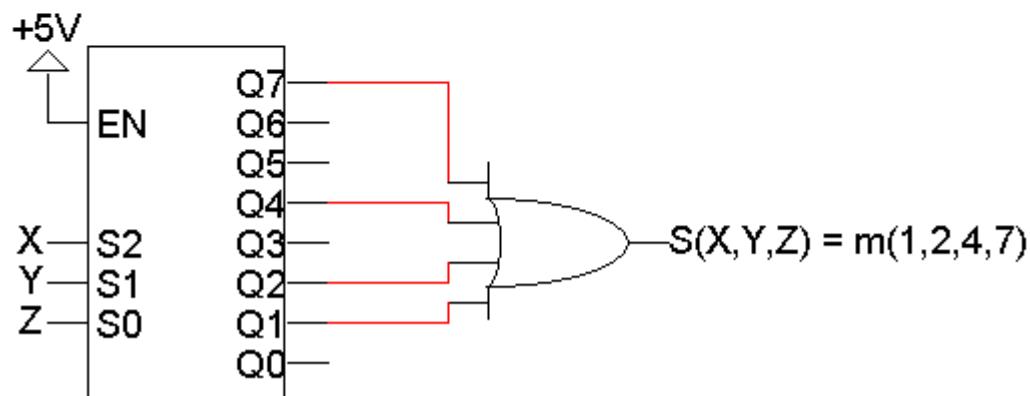
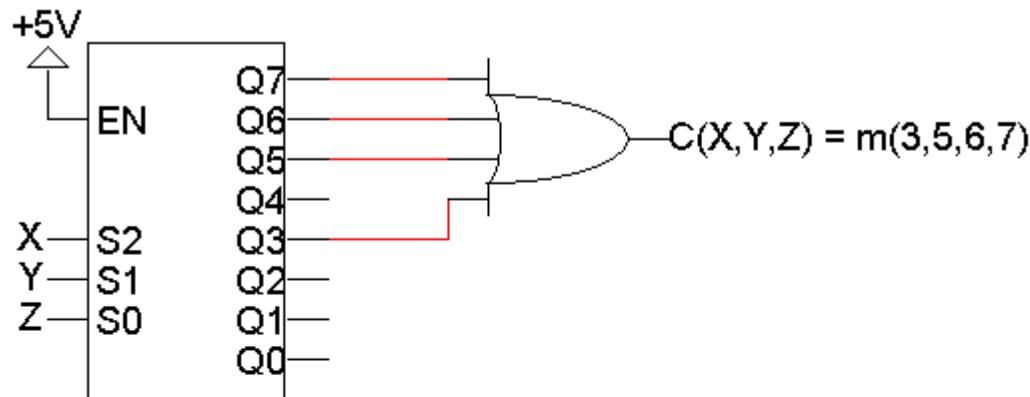
$$C(X,Y,Z) = \Sigma m(3,5,6,7)$$

$$S(X,Y,Z) = \Sigma m(1,2,4,7)$$

$$\longleftarrow 1 + 1 + 1 = 11$$

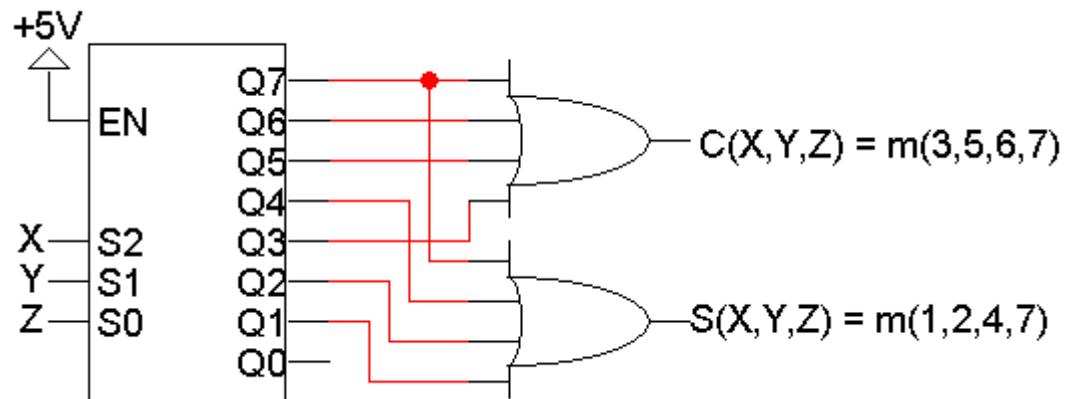
Decoder-based Adder

- Here, two 3-to-8 decoders implement C and S as sums of minterms.



Using just one decoder

- Since the two functions C and S both have the same inputs, we could use just one decoder instead of two.



Building a 3-to-8 decoder

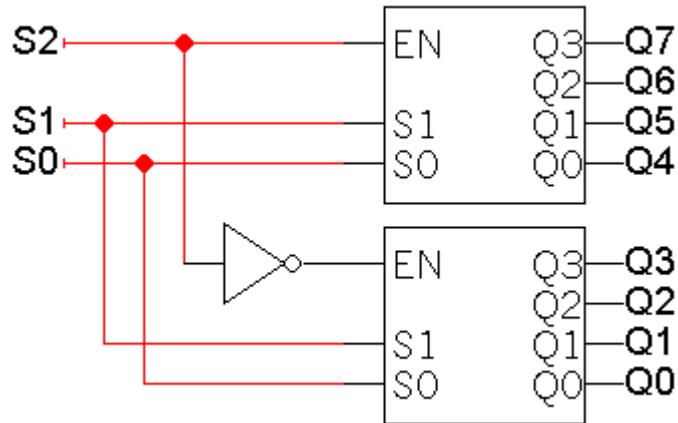
- You could build a 3-to-8 decoder directly from the truth table and equations below, just like how we built the 2-to-4 decoder
- Another way to design a decoder is to break it into smaller pieces
- Notice some patterns in the table below:
 - When $S_2 = 0$, outputs Q_0-Q_3 are generated as in a 2-to-4 decoder
 - When $S_2 = 1$, outputs Q_4-Q_7 are generated as in a 2-to-4 decoder

S_2	S_1	S_0	Q_0	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

$$\begin{aligned}Q_0 &= S_2' S_1' S_0' = m_0 \\Q_1 &= S_2' S_1' S_0 = m_1 \\Q_2 &= S_2' S_1 S_0' = m_2 \\Q_3 &= S_2' S_1 S_0 = m_3 \\Q_4 &= S_2 S_1' S_0' = m_4 \\Q_5 &= S_2 S_1' S_0 = m_5 \\Q_6 &= S_2 S_1 S_0' = m_6 \\Q_7 &= S_2 S_1 S_0 = m_7\end{aligned}$$

Decoder Expansion

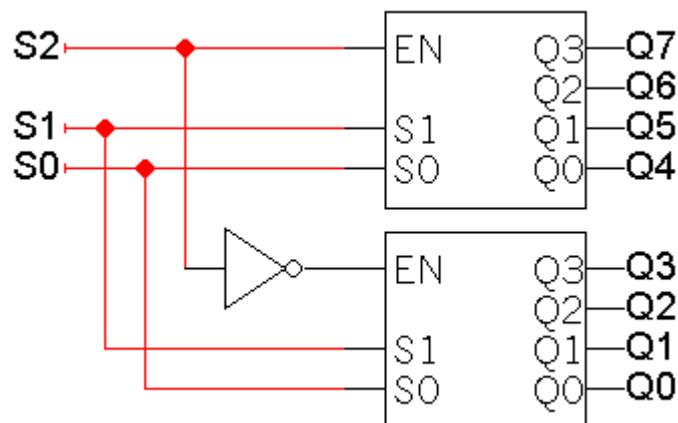
- You can use enable inputs to string decoders together. Here's a 3-to-8 decoder constructed from two 2-to-4 decoders:



S2	S1	S0	Q0	Q1	Q2	Q3	Q4	Q5	Q6	Q7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

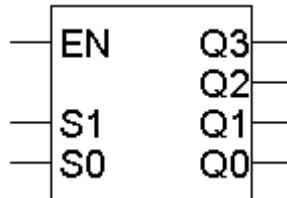
Modularity

- Be careful not to confuse the “inner” inputs and outputs of the 2-to-4 decoders with the “outer” inputs and outputs of the 3-to-8 decoder (which are in boldface)
- This is similar to having several functions in a program which all use a formal parameter “ x ”



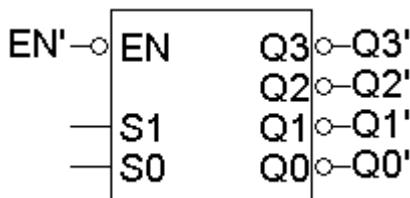
A variation of the standard decoder

- The decoders we've seen so far are **active-high** decoders



EN	S1	S0	Q0	Q1	Q2	Q3
0	x	x	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1

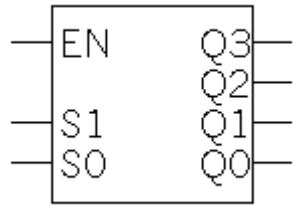
- An **active-low decoder** is the same thing, but with an inverted EN input and inverted outputs



EN'	S1'	S0'	Q0'	Q1'	Q2'	Q3'
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0
1	x	x	1	1	1	1

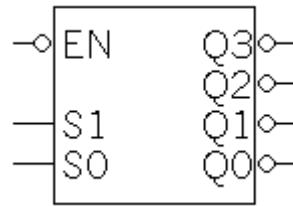
Separated at birth?

- Active-high decoders generate *minterms*, as we've already seen



$$\begin{aligned}Q_3 &= S_1 \ S_0 \\Q_2 &= S_1 \ S_0' \\Q_1 &= S_1' \ S_0 \\Q_0 &= S_1' \ S_0'\end{aligned}$$

- The output equations for an active-low decoder are mysteriously similar, yet somehow different

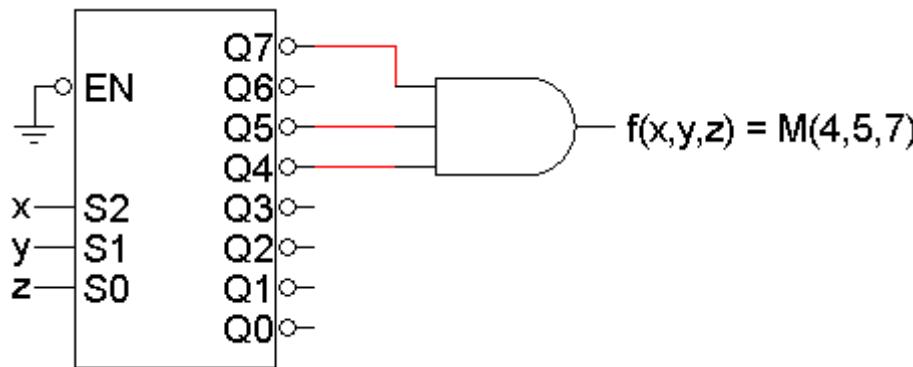


$$\begin{aligned}Q_3' &= (S_1 \ S_0)' = S_1' + S_0' \\Q_2' &= (S_1 \ S_0')' = S_1' + S_0 \\Q_1' &= (S_1' \ S_0)' = S_1 + S_0' \\Q_0' &= (S_1' \ S_0')' = S_1 + S_0\end{aligned}$$

- It turns out that active-low decoders generate *maxterms*

Active-low Decoder

- So we can use active-low decoders to implement arbitrary functions too, but as a product of maxterms
- For example, here is an implementation of the function $f(x,y,z) = \prod M(4,5,7)$ using an active-low decoder



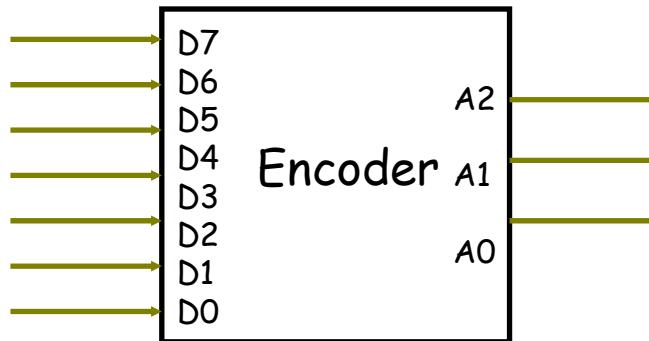
- The “ground” symbol connected to EN represents logical 0, so this decoder is always enabled

Summary of Decoders

- A n -to- 2^n decoder generates the minterms of an n -variable function
 - As such, decoders can be used to implement arbitrary functions
 - Later on we'll see other uses for decoders too
- Some variations of the basic decoder include:
 - Adding an enable input
 - Using active-low inputs and outputs to generate maxterms
- We also talked about:
 - Applying our circuit analysis and design techniques to understand and work with decoders
 - Using block symbols to encapsulate common circuits like decoders
 - Building larger decoders from smaller ones

Encoder

- An encoder is a digital function that performs the inverse operation of a decoder
- **Octal-to-Binary Encoder:** This encoder has eight inputs, one for each of the octal digits, and three outputs that generate the corresponding binary number
 - Only one input can have the value of 1 at any given time



Octal-to-Binary Encoder

□ TABLE 4-4
Truth Table for Octal-to-Binary Encoder

Inputs								Outputs		
D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀	A ₂	A ₁	A ₀
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

$$A_0 = D_1 + D_3 + D_5 + D_7,$$

$$A_2 = D_4 + D_5 + D_6 + D_7$$

$$A_1 = D_2 + D_3 + D_6 + D_7$$

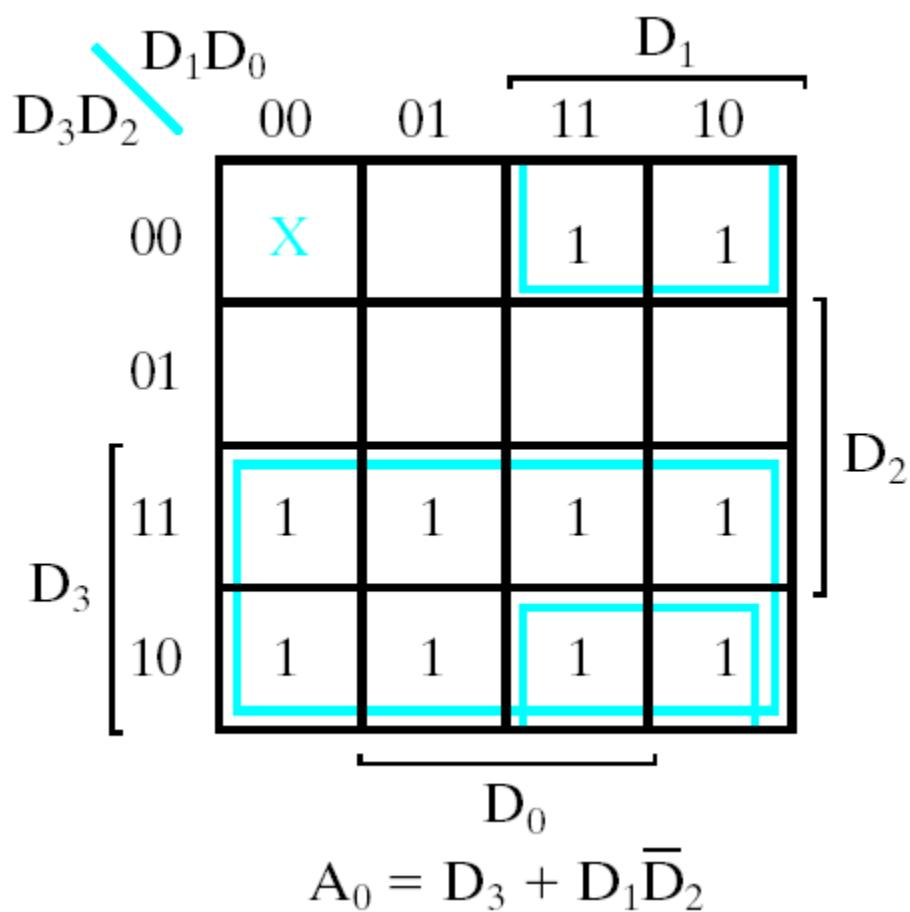
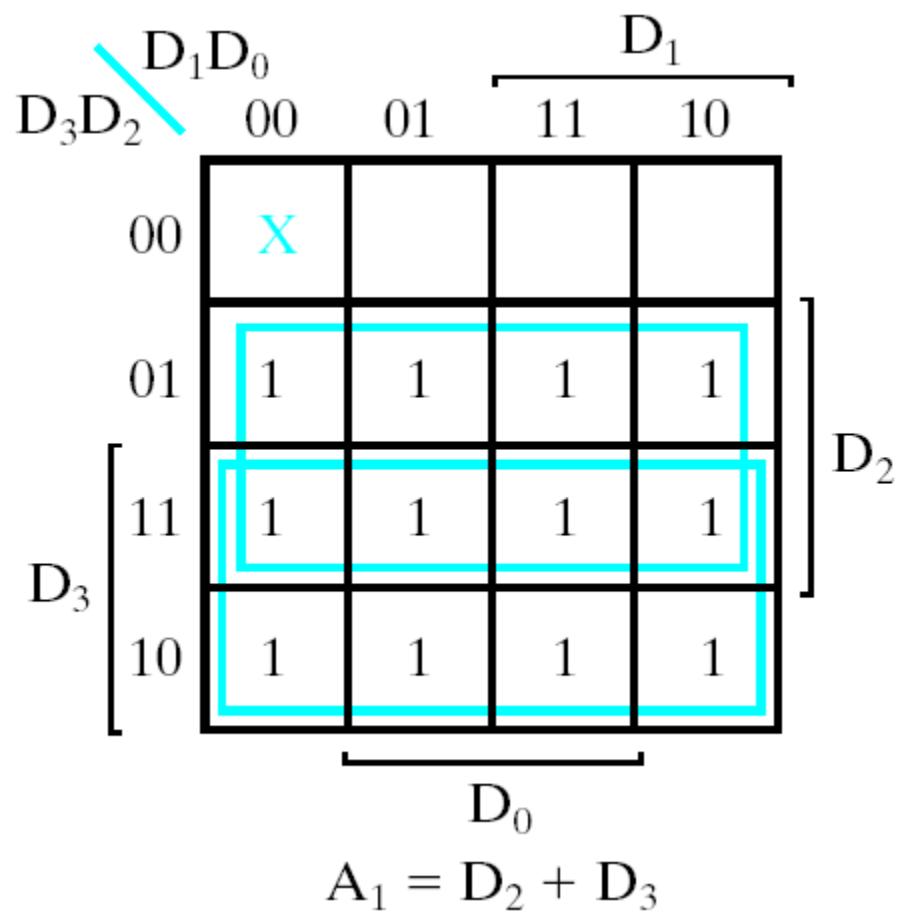
Priority Encoder

- A priority encoder is a combinational circuit that implements a priority function
- Octal-to-Binary: If more than one input is active simultaneously, the output produces an incorrect combination
- Priority Encoder: If two or more inputs are active simultaneously, the input having the highest priority takes precedence

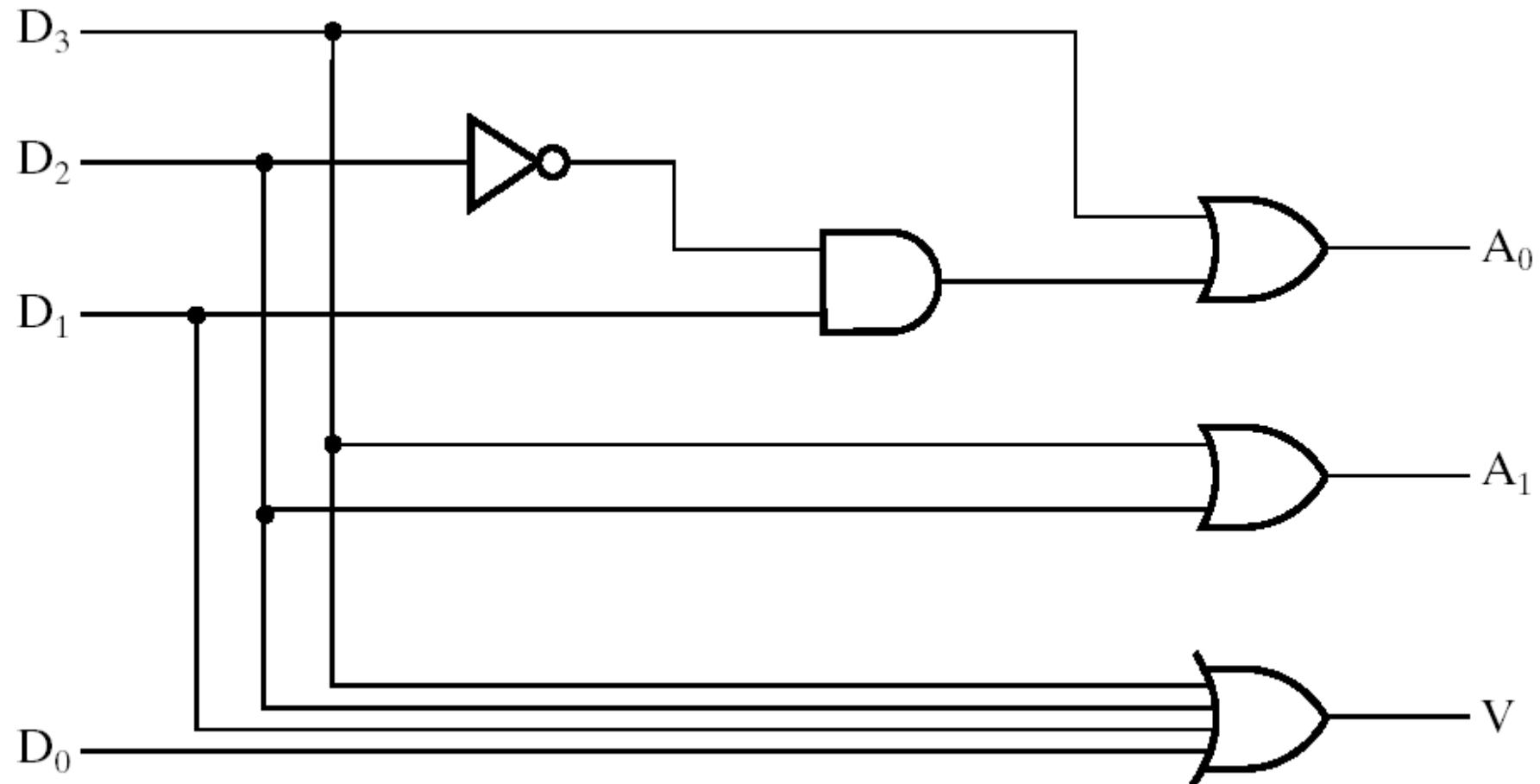
□ **TABLE 4-5**
Truth Table of Priority Encoder

Inputs				Outputs		
D ₃	D ₂	D ₁	D ₀	A ₁	A ₀	V
0	0	0	0	X	X	0
0	0	0	1	0	0	1
0	0	1	X	0	1	1
0	1	X	X	1	0	1
1	X	X	X	1	1	1

4-Input Priority Encoder

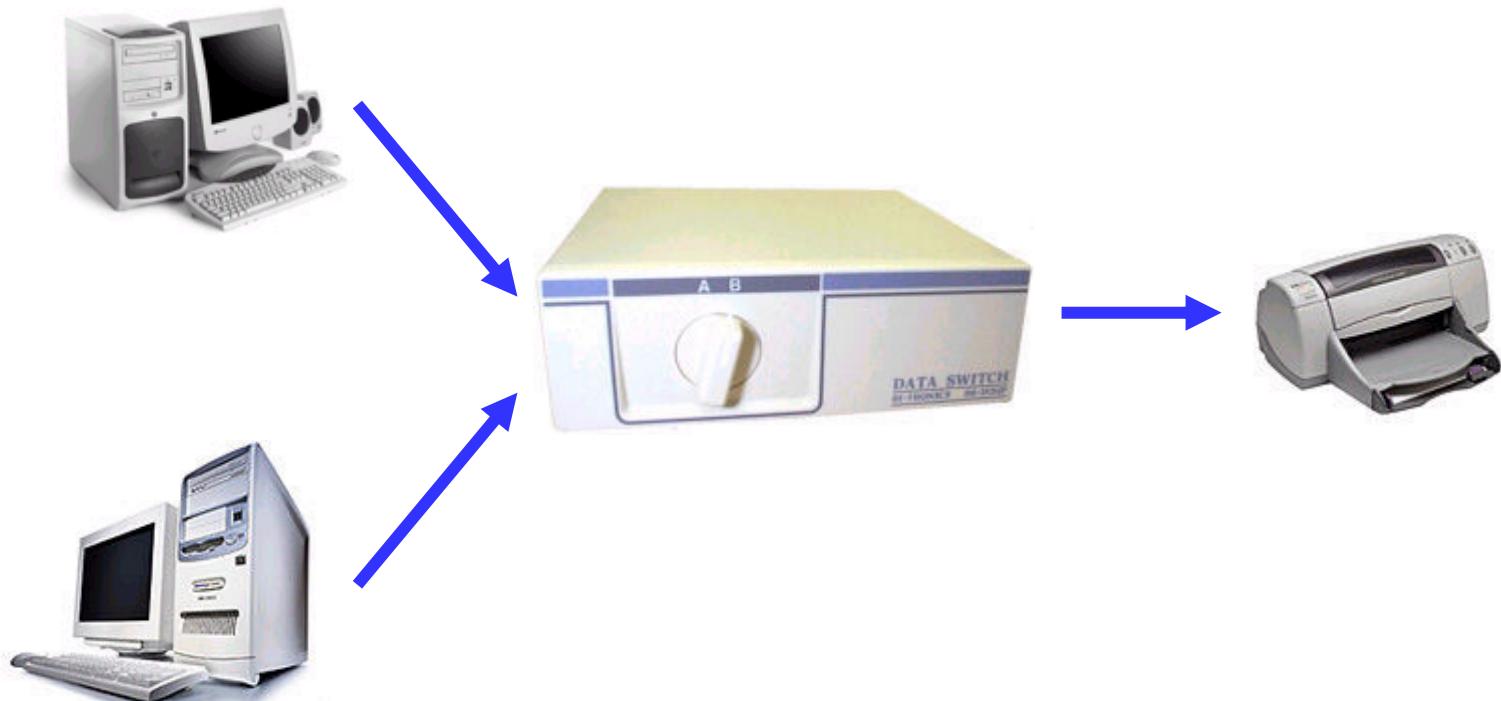


4-Input Priority Encoder



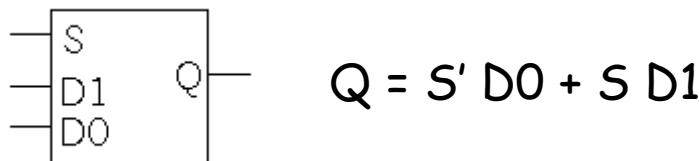
Multiplexers

- Multiplexers, or muxes, are used to choose between resources
- A real-life example: in the old days before networking, several computers could share one printer through the use of a switch.



Multiplexers

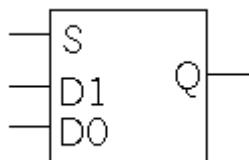
- A 2^n -to-1 multiplexer sends one of 2^n input lines to a single output line
 - A multiplexer has two sets of *inputs*:
 - 2^n **data input** lines
 - n **select** lines, to pick one of the 2^n data inputs
 - The mux *output* is a single bit, which is one of the 2^n data inputs
- The simplest example is a 2-to-1 mux:



- The select bit S controls which of the data bits D_0-D_1 is chosen:
 - If $S=0$, then D_0 is the output ($Q=D_0$).
 - If $S=1$, then D_1 is the output ($Q=D_1$).

More truth table abbreviations

- Here is a full truth table for this 2-to-1 mux, based on the equation:



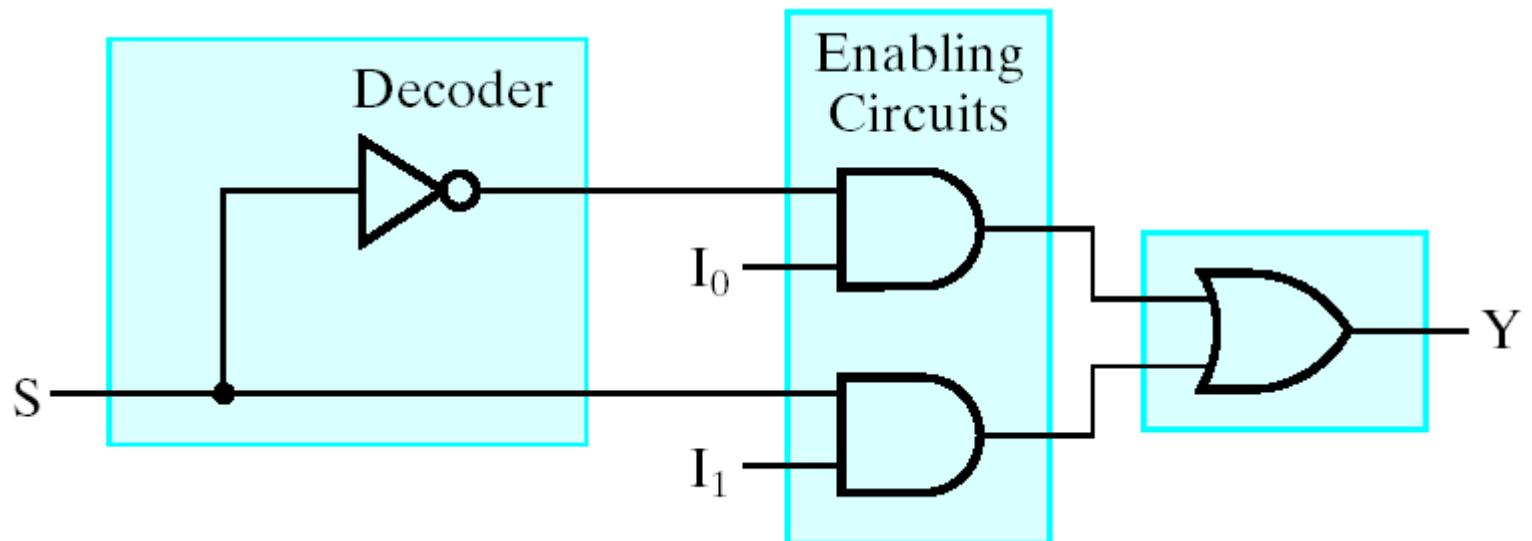
$$Q = S' D0 + S D1$$

S	D1	D0	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

- Here is another kind of abbreviated truth table
 - Input variables appear in the output column
 - This table implies that when $S=0$, the output $Q=D0$, and when $S=1$ the output $Q=D1$
 - This is a pretty close match to the equation

S	Q
0	D0
1	D1

2-to-1 Mux

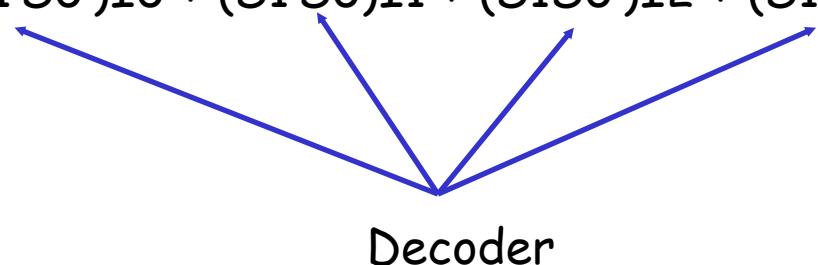


4-to-1 Mux

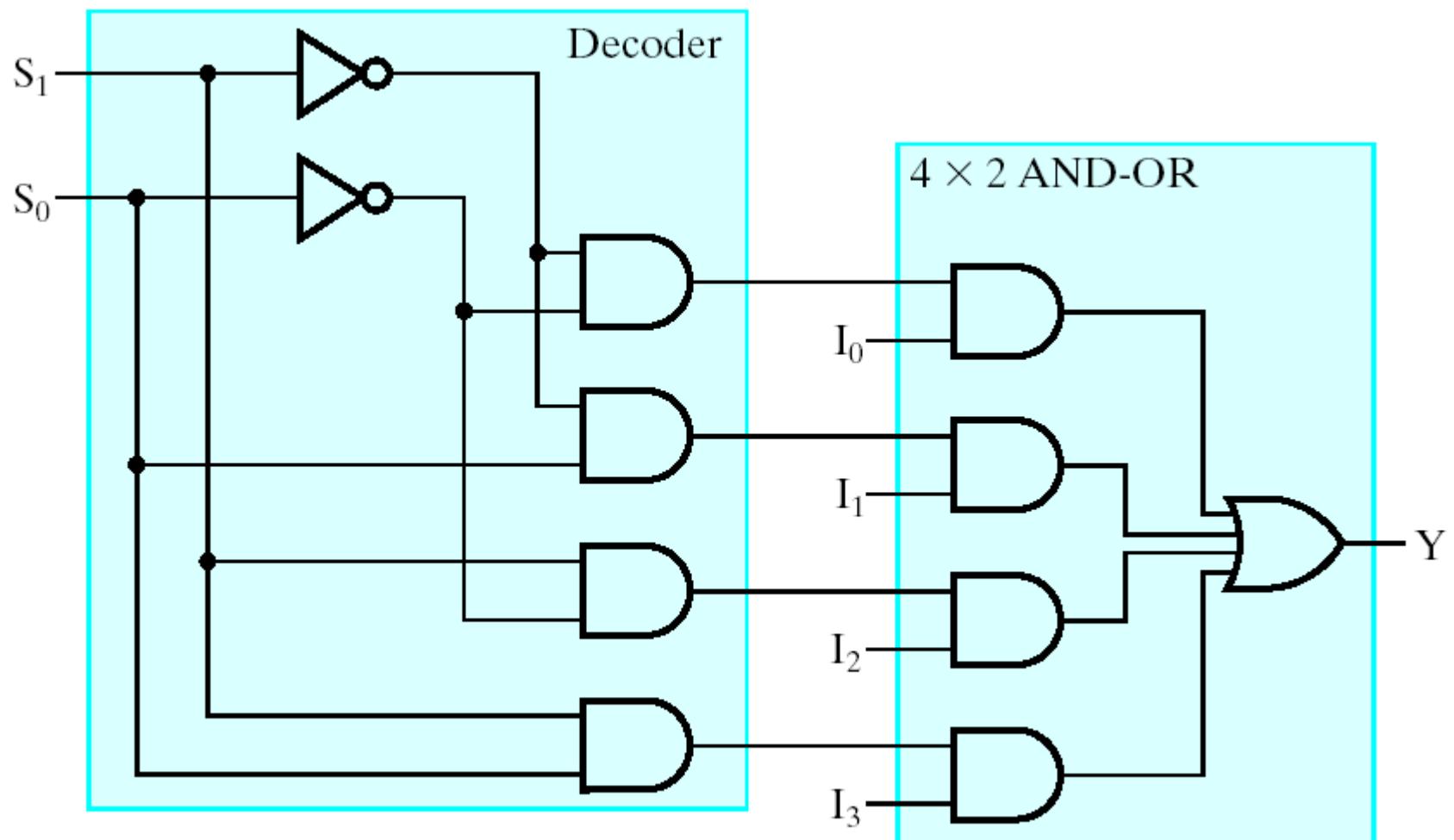
□ TABLE 4-7
Condensed Truth Table for 4-to-1-Line Multiplexer

S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

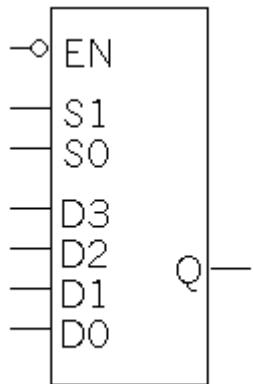
$$Y = (S_1'S_0')I_0 + (S_1'S_0)I_1 + (S_1S_0')I_2 + (S_1S_0)I_3$$



4-to-1 Mux



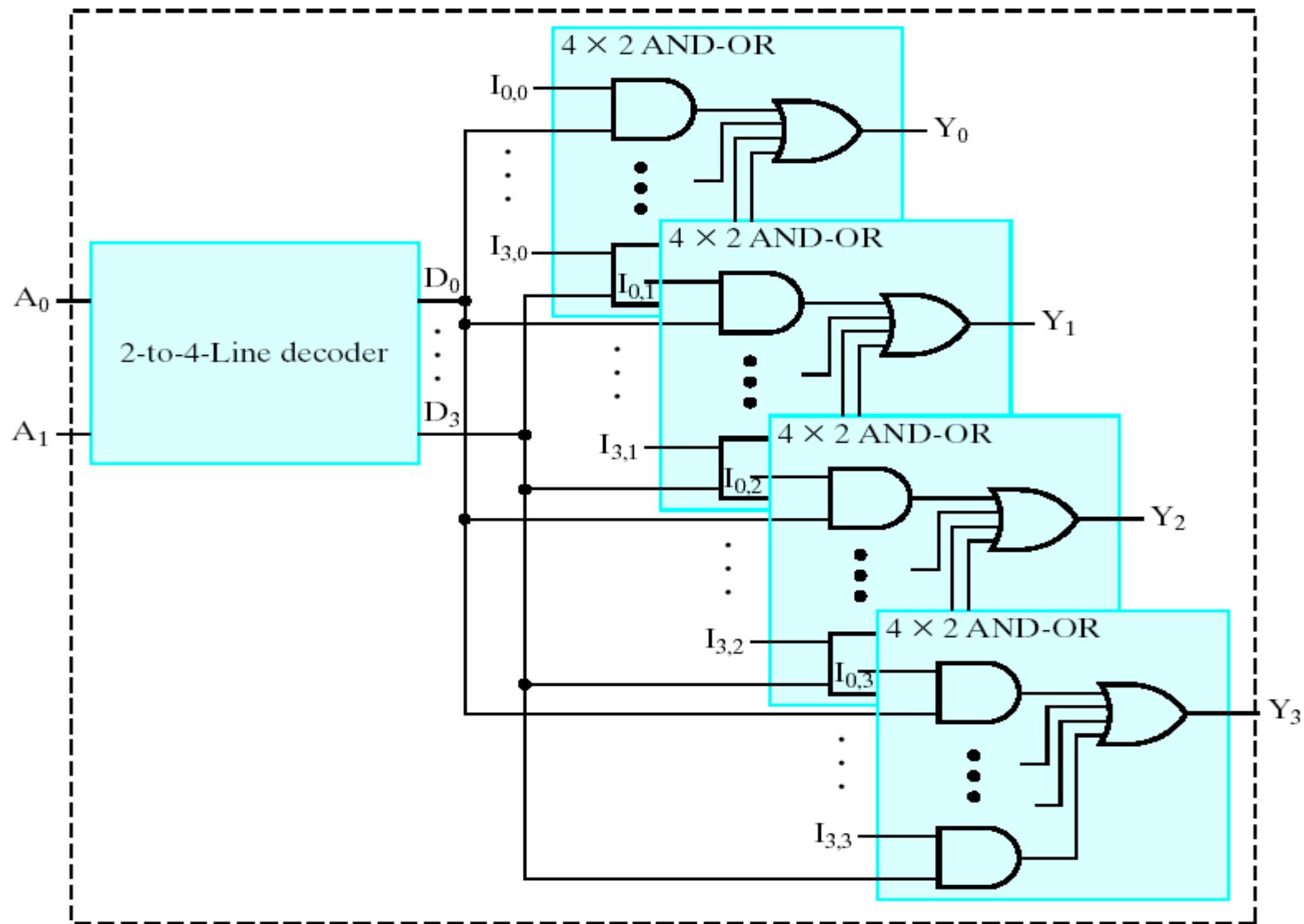
4-to-1 Mux with Enable Input



EN	S1	S0	Q
0	0	0	D0
0	0	1	D1
0	1	0	D2
0	1	1	D3
1	x	x	1

$$\begin{aligned} Q = & \overline{EN}'S1' S0' D0 + \overline{EN}'S1' S0 D1 + \overline{EN}'S1 S0' D2 \\ & + \overline{EN}'S1 S0 D3 + EN \end{aligned}$$

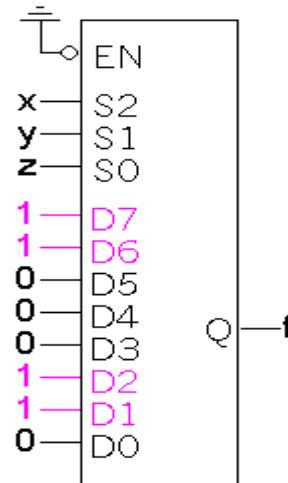
Quad 4-to-1 Mux



Implementing functions with multiplexers

- Muxes can be used to implement arbitrary functions
- One way to implement a function of n variables is to use an 2^n -to-1 mux:
 - For each minterm m_i of the function, connect 1 to mux data input D_i .
Each data input corresponds to one row of the truth table
 - Connect the function's input variables to the mux select inputs. These are used to indicate a particular input combination
- For example, let's look at $f(x,y,z) = \Sigma m(1,2,6,7)$.

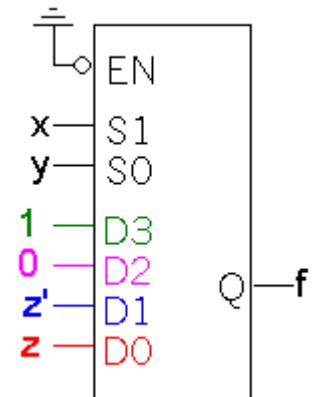
x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



A more efficient way

- We can actually implement $f(x,y,z) = \sum m(1,2,6,7)$ with just a 4-to-1 mux, instead of an 8-to-1
- Step 1: Find the truth table for the function, and group the rows into pairs. Within each pair of rows, x and y are the same, so f is a function of z only.
 - When $xy=00$, $f=z$
 - When $xy=01$, $f=z'$
 - When $xy=10$, $f=0$
 - When $xy=11$, $f=1$
- Step 2: Connect the first two input variables of the truth table (here, x and y) to the select bits $S1$ $S0$ of the 4-to-1 mux.
- Step 3: Connect the equations above for $f(z)$ to the data inputs $D0-D3$.

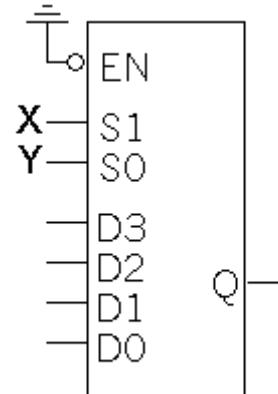
x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



Example: multiplexer-based adder

- Let's implement the adder carry function, $C(X,Y,Z)$, with muxes
- There are three inputs, so we'll need a 4-to-1 mux
- The basic setup is to connect two of the input variables (usually the first two in the truth table) to the mux select inputs

X	Y	Z	C
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



With $S1=X$ and $S0=Y$, then

$$Q = X'Y'D0 + X'YD1 + XY'D2 + XYD3$$

Multiplexer-based carry

- We can set the multiplexer data inputs D0-D3, by fixing X and Y and finding equations for C in terms of just Z.

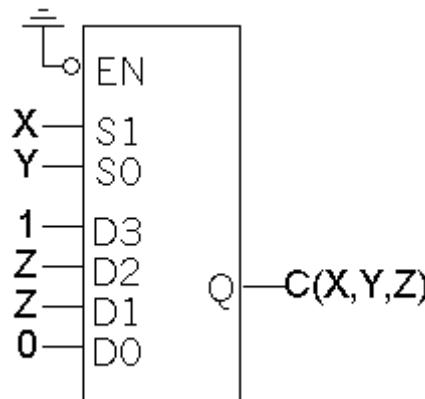
X	Y	Z	C
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

When XY=00, C=0

When XY=01, C=Z

When XY=10, C=Z

When XY=11, C=1



$$\begin{aligned}C &= X' Y' D0 + X' Y D1 + X Y' D2 + X Y D3 \\&= X' Y' 0 + X' Y Z + X Y' Z + X Y 1 \\&= X' Y Z + X Y' Z + X Y \\&= \Sigma m(3, 5, 6, 7)\end{aligned}$$

Multiplexer-based sum

- Here's the same thing, but for the sum function $S(X,Y,Z)$

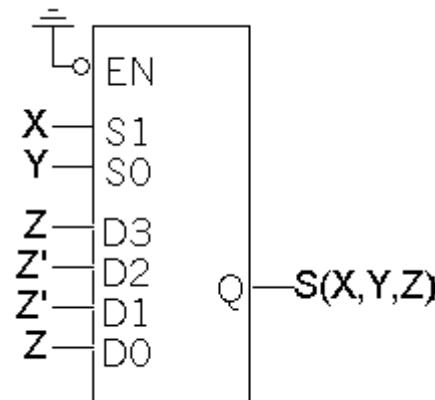
X	Y	Z	S
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

When XY=00, S=Z

When XY=01, S=Z'

When XY=10, S=Z'

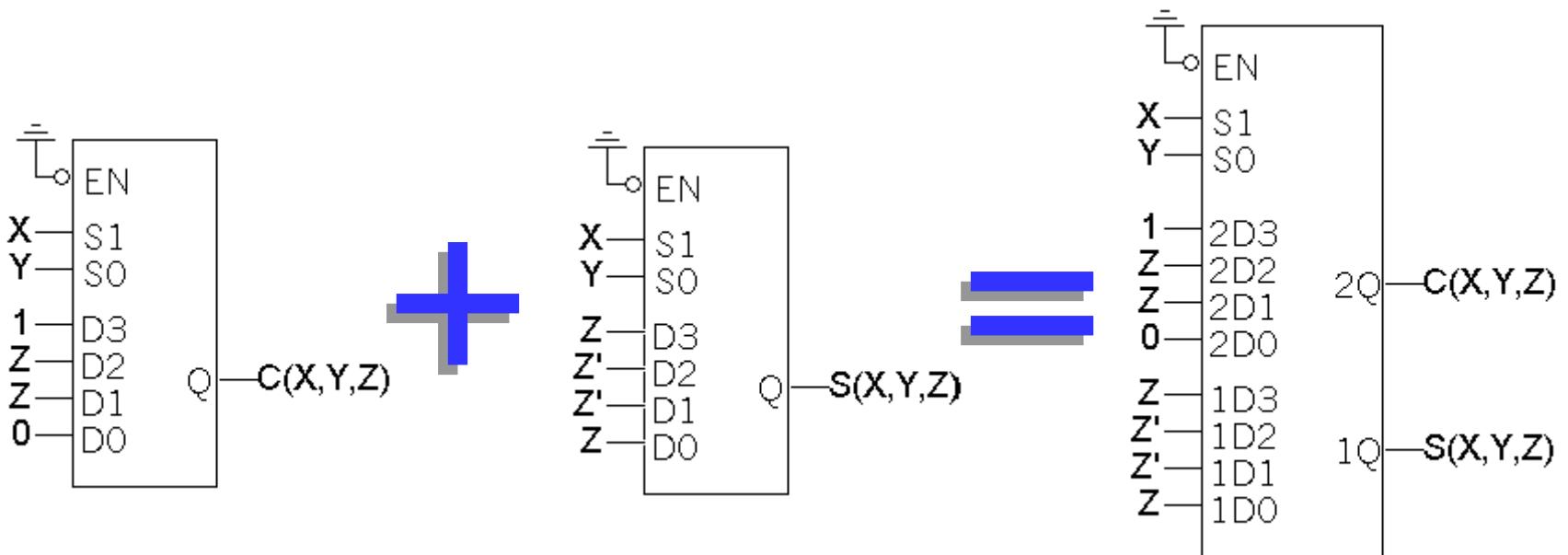
When XY=11, S=Z



$$\begin{aligned}S &= X' Y' D0 + X' Y D1 + X Y' D2 + X Y D3 \\&= X' Y' Z + X' Y Z' + X Y' Z' + X Y Z \\&= \Sigma m(1, 2, 4, 7)\end{aligned}$$

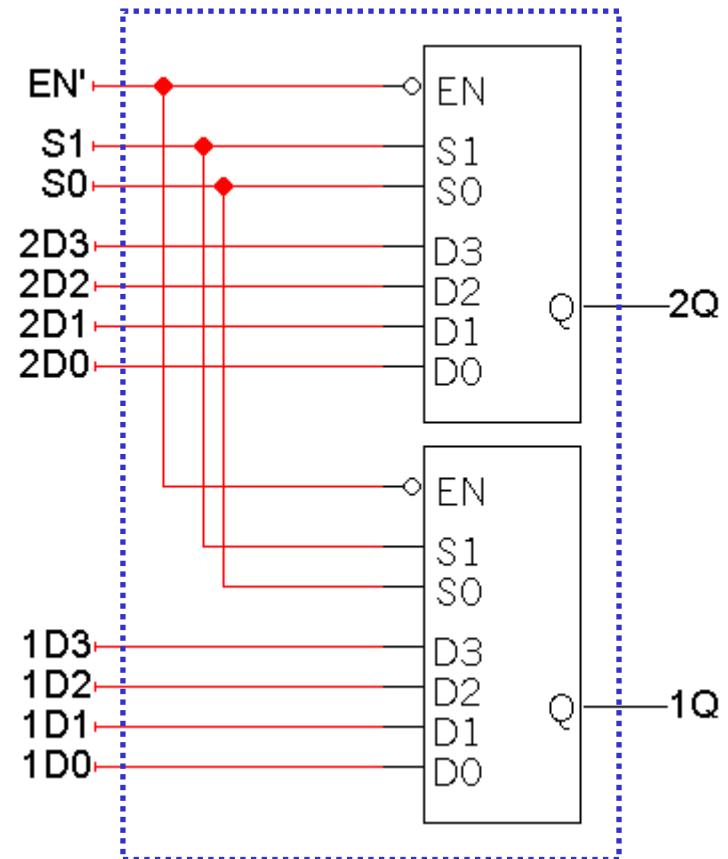
Dual multiplexer-based full adder

- We need *two* separate 4-to-1 muxes: one for C and one for S
- But sometimes it's convenient to think about the adder output as being a single 2-bit number, instead of as two separate functions
- A **dual 4-to-1 mux** gives the illusion of 2-bit data inputs and outputs
 - It's really just two 4-to-1 muxes connected together

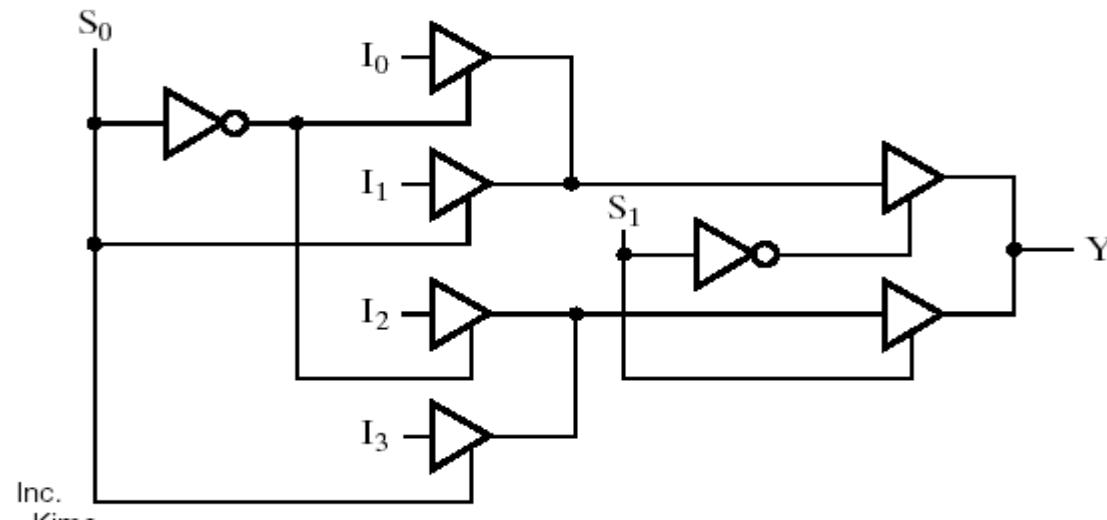
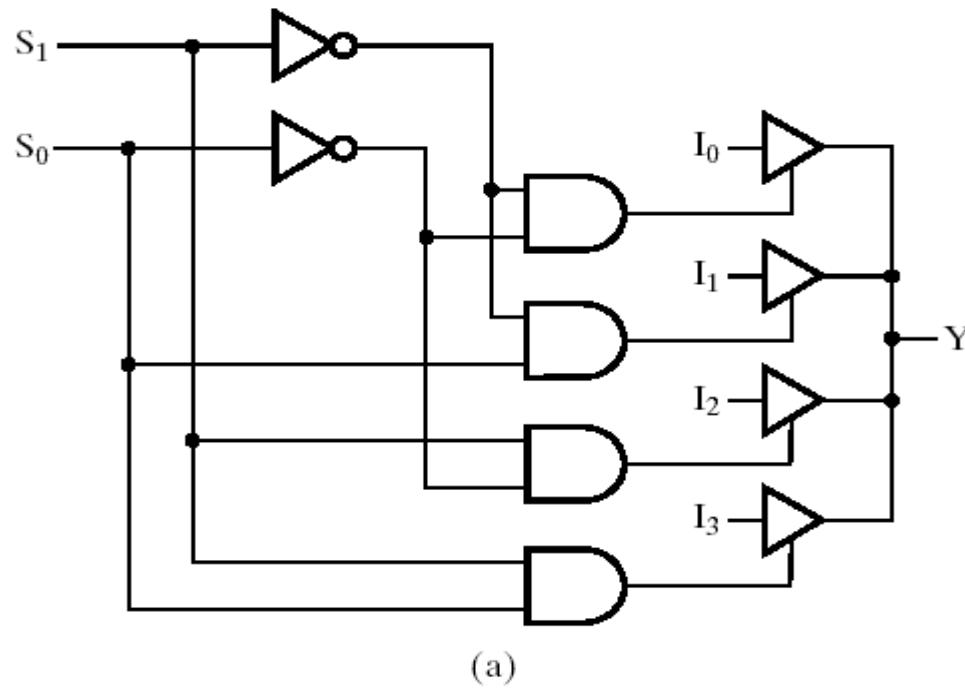


Dual muxes in more detail

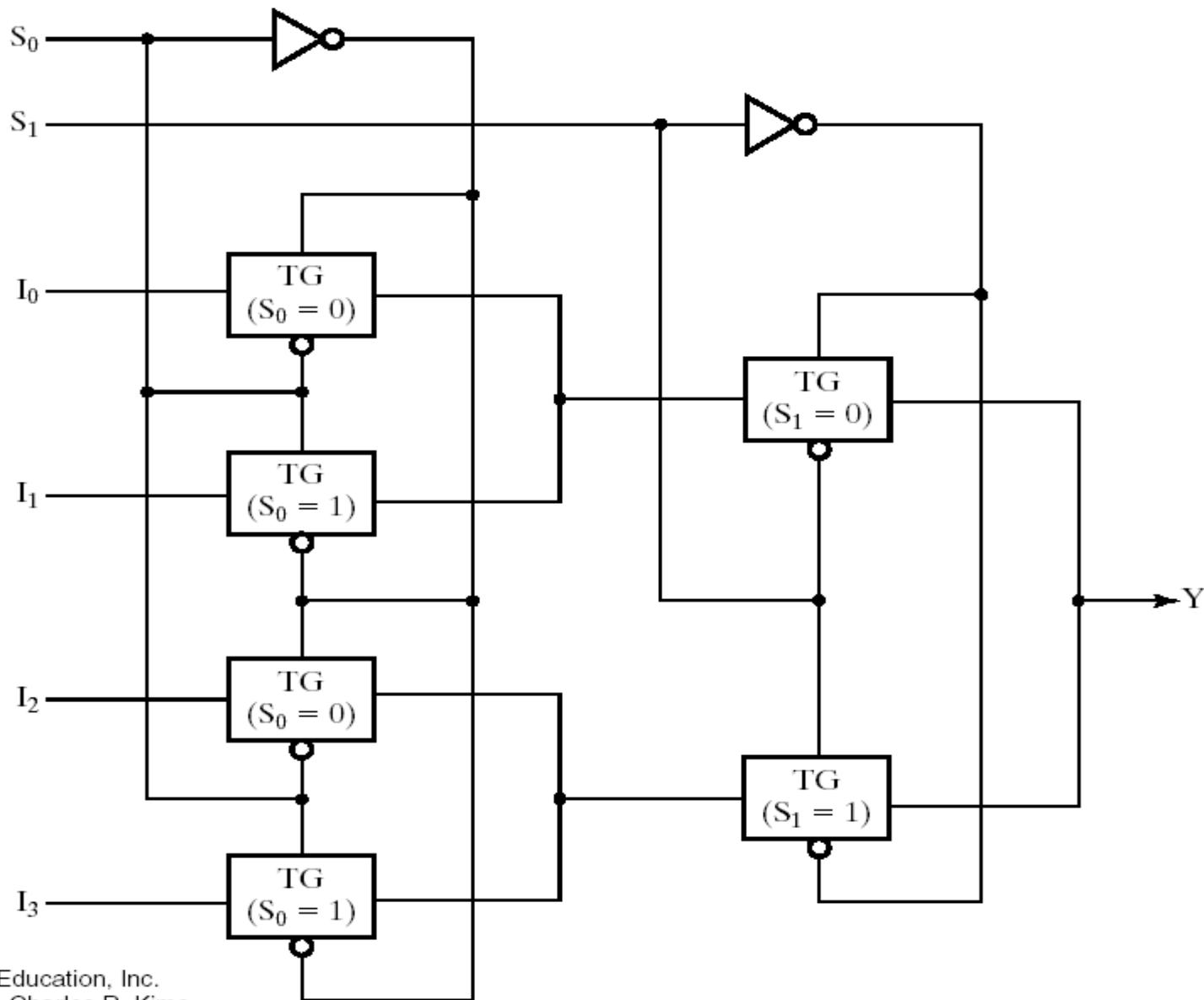
- You can make a dual 4-to-1 mux by connecting two 4-to-1 muxes. ("Dual" means "two-bit values.")
- In the diagram on the right, we're using S1-S0 to choose one of the following *pairs* of inputs:
 - 2D3 1D3, when S1 S0 = 11
 - 2D2 1D2, when S1 S0 = 10
 - 2D1 1D1, when S1 S0 = 01
 - 2D0 1D0, when S1 S0 = 00



Selecting based on 3-state buffers



Selecting based on TGs



Summary of Muxes

- A 2^n -to-1 multiplexer routes one of 2^n input lines to a single output line
- Just like decoders,
 - Muxes are common enough to be supplied as stand-alone devices for use in modular designs.
 - Muxes can implement arbitrary functions
- We saw some variations of the standard multiplexer:
 - Smaller muxes can be combined to produce larger ones
 - We can add active-low or active-high enable inputs
- As always, we use truth tables and Boolean algebra to analyze things

Chapter 5: Arithmetic Functions and Circuits

Binary Addition by Hand

- You can add two binary numbers one column at a time starting from the right, just as you add two decimal numbers
- But remember that it's binary. For example, $1 + 1 = 10$ and you have to carry!

The initial carry
in is implicitly 0



$$\begin{array}{r} & 1 & 1 & 1 & 0 \\ & 1 & 0 & 1 & 1 \\ + & 1 & 1 & 1 & 0 \\ \hline & 1 & 1 & 0 & 0 & 1 \end{array} \begin{array}{l} \text{Carry in} \\ \text{Augend} \\ \text{Addend} \\ \text{Sum} \end{array}$$



most significant
bit, or MSB



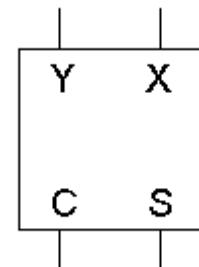
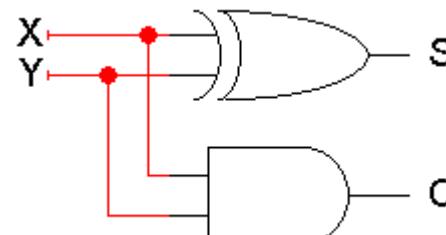
least significant
bit, or LSB

Adding Two Bits

- A hardware adder by copying the human addition algorithm
- **Half adder:** Adds two bits and produces a two-bit result: a **sum** (the right bit) and a **carry out** (the left bit)
- Here are truth tables, equations, circuit and block symbol

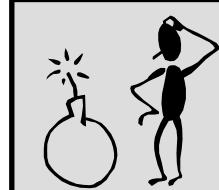
X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$\begin{aligned}0 + 0 &= 0 \\0 + 1 &= 1 \\1 + 0 &= 1 \\1 + 1 &= 10\end{aligned}$$



$$C = XY$$

$$\begin{aligned}S &= X' Y + X Y' \\&= X \oplus Y\end{aligned}$$



Be careful! Now we're using **+** for both arithmetic addition and the logical OR operation.

Adding Three Bits

- But what we really need to do is add *three* bits: the augend and addend, and the *carry in* from the right.

$$\begin{array}{r} & 1 & 1 & 1 & 0 \\ & 1 & 0 & & 1 \\ + & 1 & 1 & 1 & 1 \\ \hline & 1 & 1 & 0 & 1 \end{array}$$

X	Y	C_{in}	C_{out}	S	
0	0	0	0	0	$0 + 0 + 0 = 00$
0	0	1	0	1	$0 + 0 + 0 = 01$
0	1	0	0	1	$0 + 1 + 0 = 01$
0	1	1	1	0	$0 + 1 + 1 = 10$
1	0	0	0	1	$1 + 0 + 0 = 01$
1	0	1	1	0	$1 + 0 + 1 = 10$
1	1	0	1	0	$1 + 1 + 0 = 10$
1	1	1	1	1	$1 + 1 + 1 = 11$

Full Adder

- **Full adder:** Three bits of input, two-bit output consisting of a sum and a carry out
- Using Boolean algebra, we get the equations shown here
 - XOR operations simplify the equations a bit
 - We used algebra because you can't easily derive XORs from K-maps

X	Y	C_{in}	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

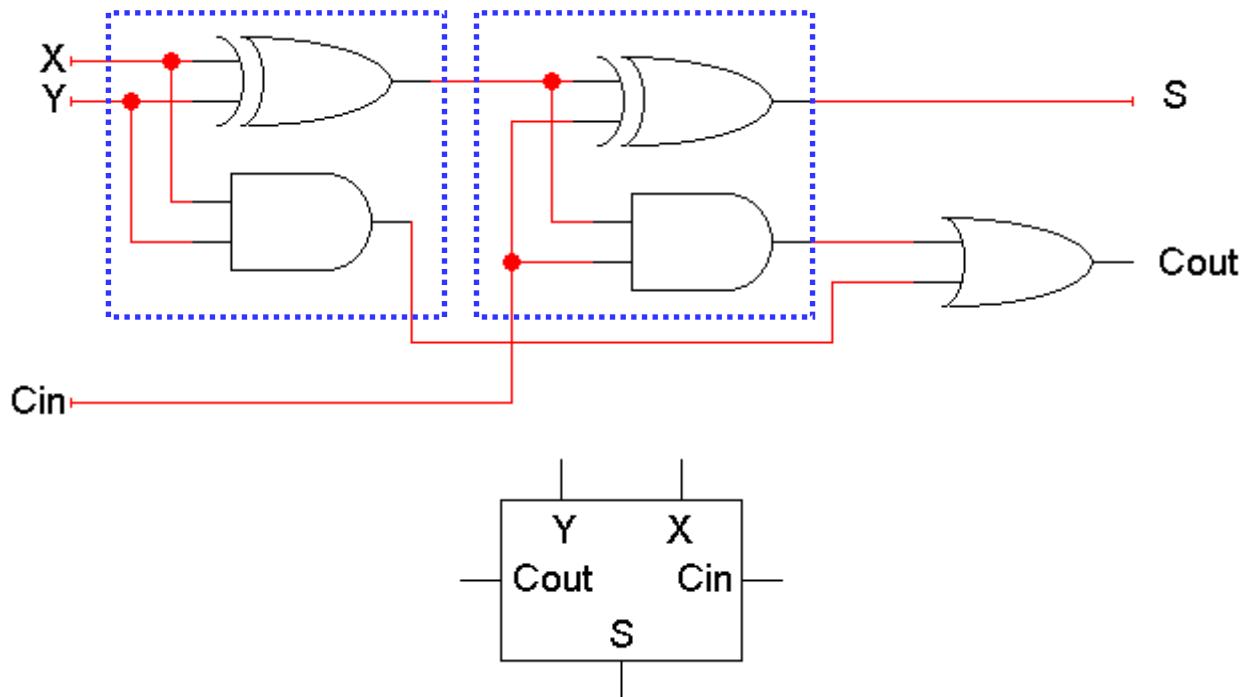
$$\begin{aligned}S &= \Sigma m(1,2,4,7) \\&= X' Y' C_{in} + X' Y C_{in}' + X Y' C_{in}' + X Y C_{in} \\&= X' (Y' C_{in} + Y C_{in}') + X (Y' C_{in}' + Y C_{in}) \\&= X' (Y \oplus C_{in}) + X (Y \oplus C_{in})' \\&= X \oplus Y \oplus C_{in}\end{aligned}$$

$$\begin{aligned}C_{out} &= \Sigma m(3,5,6,7) \\&= X' Y C_{in} + X Y' C_{in} + X Y C_{in}' + X Y C_{in} \\&= (X' Y + X Y') C_{in} + X Y (C_{in}' + C_{in}) \\&= (X \oplus Y) C_{in} + X Y\end{aligned}$$

Full Adder Circuit

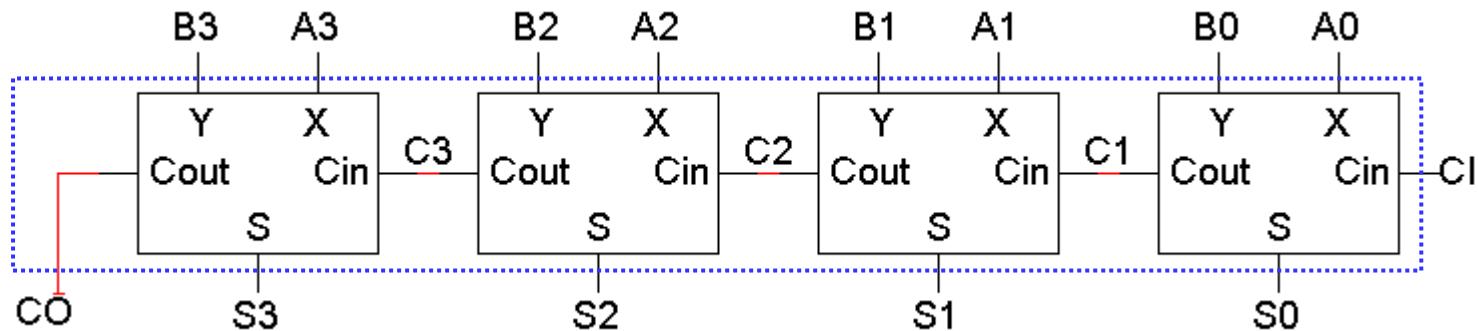
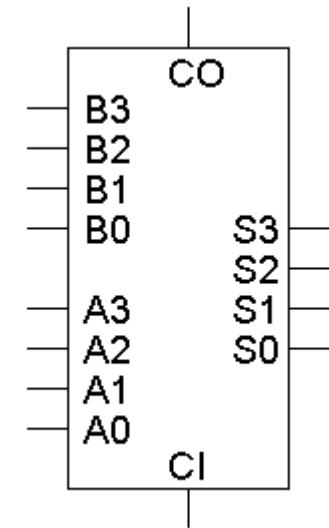
- These things are called half adders and full adders because you can build a full adder by putting together two half adders!

$$S = X \oplus Y \oplus C_{in}$$
$$C_{out} = (X \oplus Y) C_{in} + XY$$



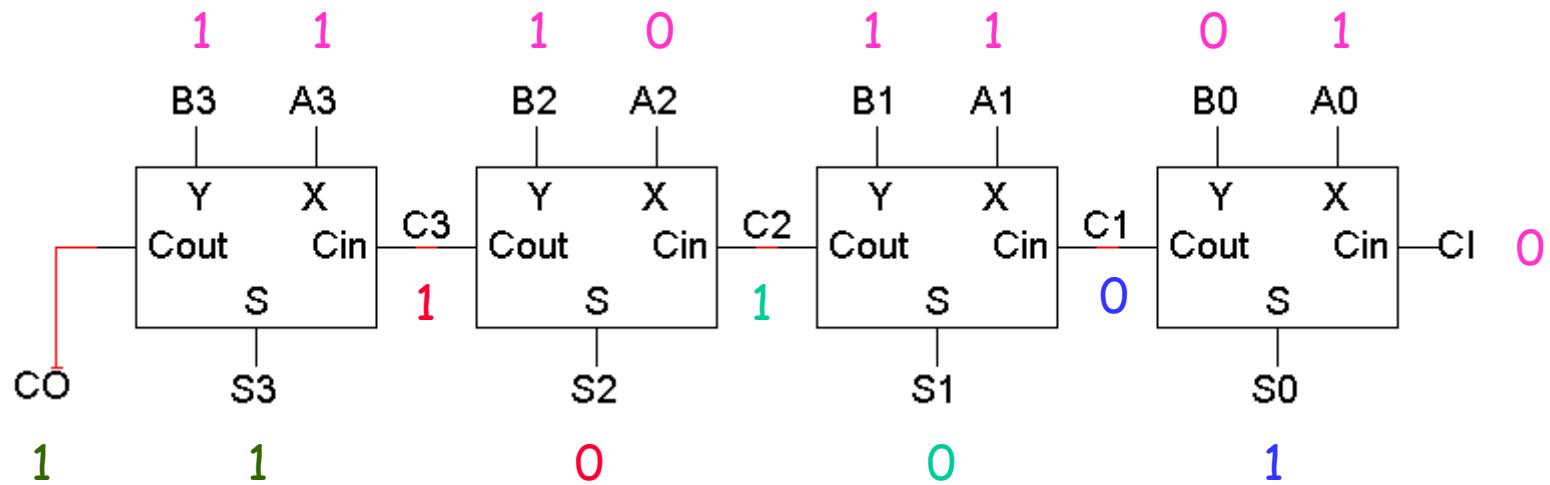
A 4-bit Adder

- Four full adders together make a 4-bit adder
- There are nine total inputs:
 - Two 4-bit numbers, $A_3 A_2 A_1 A_0$ and $B_3 B_2 B_1 B_0$
 - An initial carry in, CI
- The five outputs are:
 - A 4-bit sum, $S_3 S_2 S_1 S_0$
 - A carry out, CO
- Imagine designing a nine-input adder without this hierarchical structure—you'd have a 512-row truth table with five outputs!



An example of 4-bit addition

- Let's try our initial example: $A=1011$ (eleven), $B=1110$ (fourteen)



- Fill in all the inputs, including $CI=0$
- The circuit produces $C1$ and $S0$ ($1 + 0 + 0 = 01$)
- Use $C1$ to find $C2$ and $S1$ ($1 + 1 + 0 = 10$)
- Use $C2$ to compute $C3$ and $S2$ ($0 + 1 + 1 = 10$)
- Use $C3$ to compute CO and $S3$ ($1 + 1 + 1 = 11$)

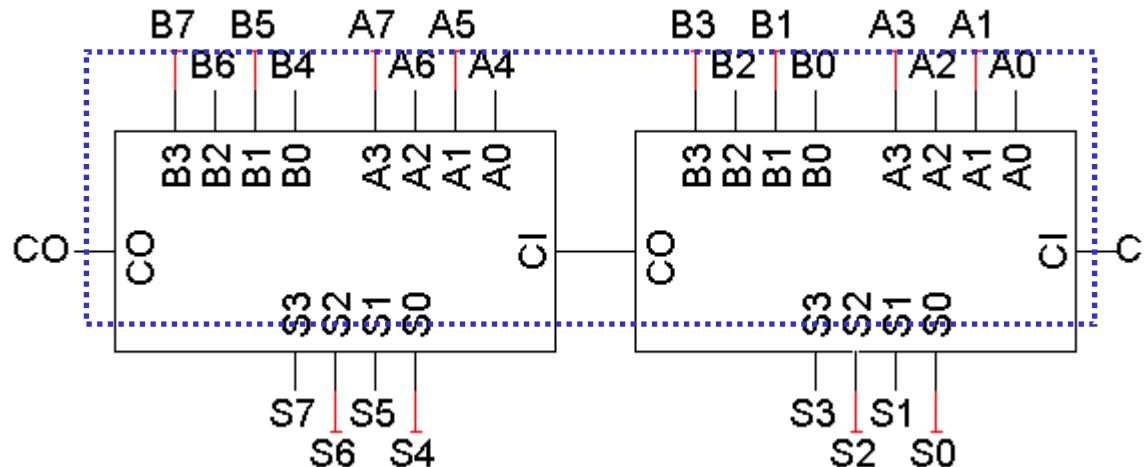
The final answer is 11001 (twenty-five)

Overflow

- In this case, note that the answer (11001) is *five* bits long, while the inputs were each only four bits (1011 and 1110). This is called **overflow**
- Although the answer 11001 is correct, we cannot use that answer in any subsequent computations with this 4-bit adder
- For **unsigned addition**, overflow occurs when the carry out is 1

Hierarchical Adder Design

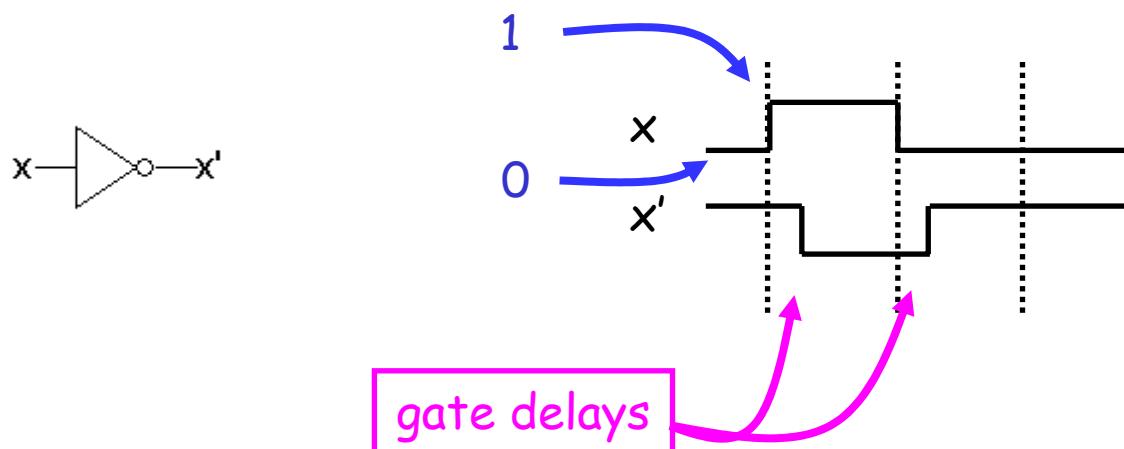
- When you add two 4-bit numbers the carry in is always 0, so why does the 4-bit adder have a CI input?
- One reason is so we can put 4-bit adders together to make even larger adders! This is just like how we put four full adders together to make the 4-bit adder in the first place
- Here is an 8-bit adder, for example.



- CI is also useful for subtraction!

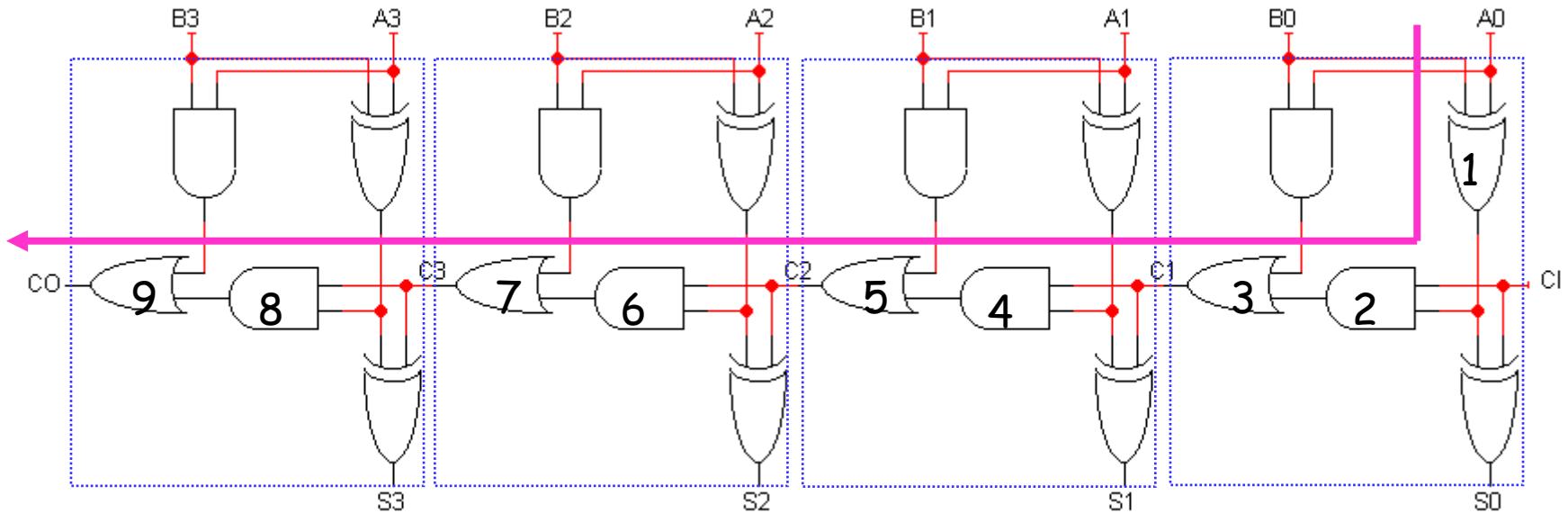
Gate Delays

- Every gate takes some small fraction of a second between the time inputs are presented and the time the correct answer appears on the outputs. This little fraction of a second is called a **gate delay**
- There are actually detailed ways of calculating gate delays that can get quite complicated, but for this class, let's just assume that there's some small constant delay that's the same for all gates
- We can use a **timing diagram** to show gate delays graphically



Delays in the Ripple Carry Adder

- The diagram below shows a 4-bit adder completely drawn out
- This is called a **ripple carry** adder, because the inputs A_0 , B_0 and CI "ripple" leftwards until CO and S_3 are produced
- Ripple carry adders are slow!
 - Our example addition with 4-bit inputs required 5 "steps"
 - There is a very long path from A_0 , B_0 and CI to CO and S_3
 - For an n -bit ripple carry adder, the longest path has $2n+1$ gates
 - Imagine a 64-bit adder. The longest path would have 129 gates!



A faster way to compute carry outs

- Instead of waiting for the carry out from all the previous stages, we could compute it directly with a two-level circuit, thus minimizing the delay
- First we define two functions
 - The “generate” function g_i produces 1 when there *must* be a carry out from position i (i.e., when A_i and B_i are both 1).

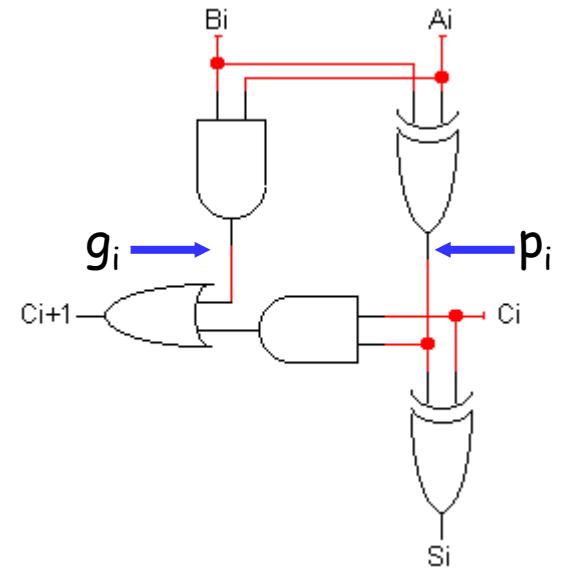
$$g_i = A_i B_i$$

- The “propagate” function p_i is true when, if there is an incoming carry, it is propagated (i.e., when $A_i=1$ or $B_i=1$, but not both).

$$p_i = A_i \oplus B_i$$

- Then we can rewrite the carry out function:

$$C_{i+1} = g_i + p_i C_i$$



A_i	B_i	C_i	C_{i+1}
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

Algebraic Carry Out

- Let's look at the carry out equations for specific bits, using the general equation from the previous page $c_{i+1} = g_i + p_i c_i$:

$$c_1 = g_0 + p_0 c_0$$

Ready to see the circuit?

$$c_2 = g_1 + p_1 c_1$$

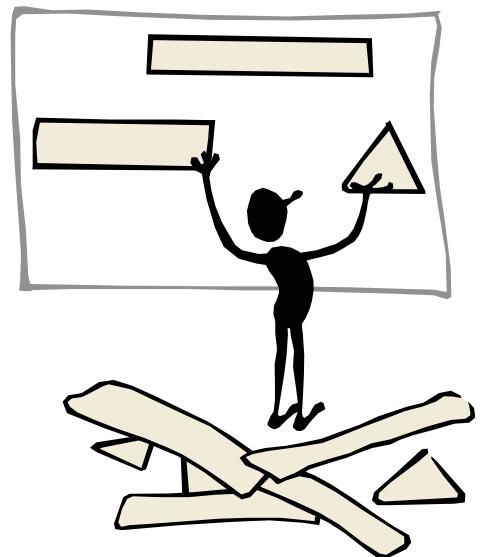
$$\begin{aligned} &= g_1 + p_1(g_0 + p_0 c_0) \\ &= g_1 + p_1 g_0 + p_1 p_0 c_0 \end{aligned}$$

$$c_3 = g_2 + p_2 c_2$$

$$\begin{aligned} &= g_2 + p_2(g_1 + p_1 g_0 + p_1 p_0 c_0) \\ &= g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0 \end{aligned}$$

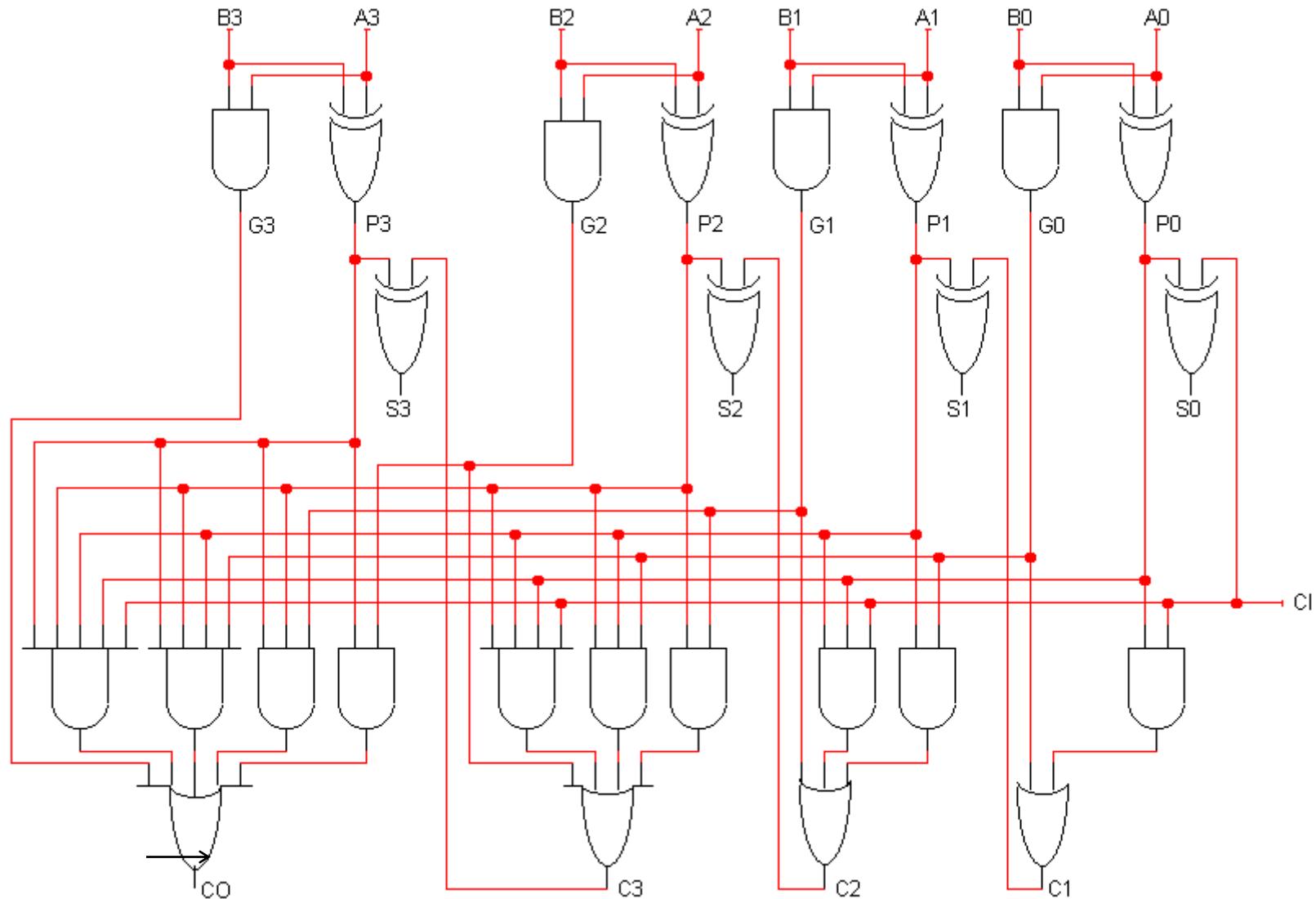
$$c_4 = g_3 + p_3 c_3$$

$$\begin{aligned} &= g_3 + p_3(g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0) \\ &= g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0 \end{aligned}$$



- These expressions are all sums of products, so we can use them to make a circuit with only a two-level delay.

A 4-bit CLA Circuit



Carry Lookahead Adders

- This is called a **carry lookahead adder**
- By adding more hardware, we reduced the number of levels in the circuit and sped things up
- We can “cascade” carry lookahead adders, just like ripple carry adders
- How much faster is this?
 - For a 4-bit adder, not much. CLA: 4 gates, RCA: 9 gates
 - But if we do the cascading properly, for a 16-bit carry lookahead adder, 8 gates vs. 33
 - Newer CPUs these days use 64-bit adders. That's 12 vs. 129 gates!
- The delay of a carry lookahead adder grows *logarithmically* with the size of the adder, while a ripple carry adder's delay grows *linearly*
- Trade-off between complexity and performance. Ripple carry adders are simpler, but slower. Carry lookahead adders are faster but more complex

Multiplication

- Multiplication can't be that hard!
 - It's just repeated addition
 - If we have adders, we can do multiplication also
- Remember that the AND operation is equivalent to multiplication on two bits:

a	b	ab
0	0	0
0	1	0
1	0	0
1	1	1

a	b	$a \times b$
0	0	0
0	1	0
1	0	0
1	1	1

Binary multiplication example

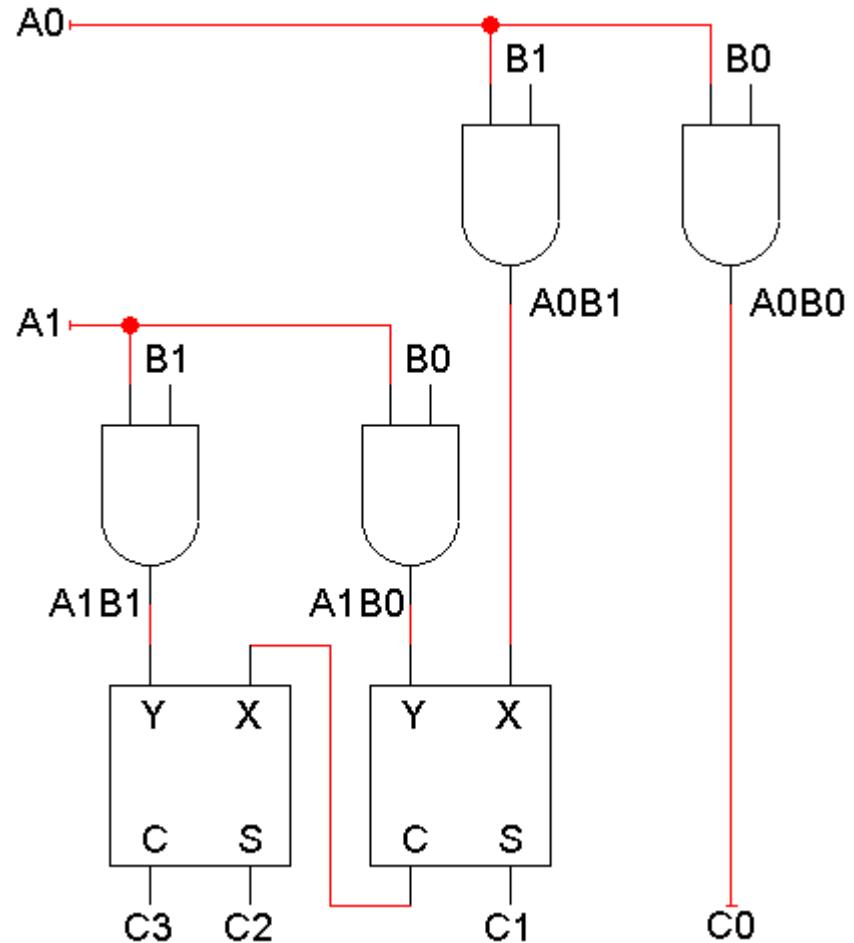
x	1	1	0	1			
	0	1	1	0			
<hr/>							
	0	0	0	0	Partial products		
	1	1	0	1			
	1	1	0	1			
+	0	0	0	0			
<hr/>					Product		
	1	0	0	1	1	1	0

- Since we always multiply by either 0 or 1, the **partial products** are always either **0000** or the multiplicand (**1101** in this example)
- There are four partial products which are added to form the result
 - We can add them in pairs, using three adders
 - Even though the product has up to 8 bits, we can use 4-bit adders if we “stagger” them leftwards, like the partial products themselves

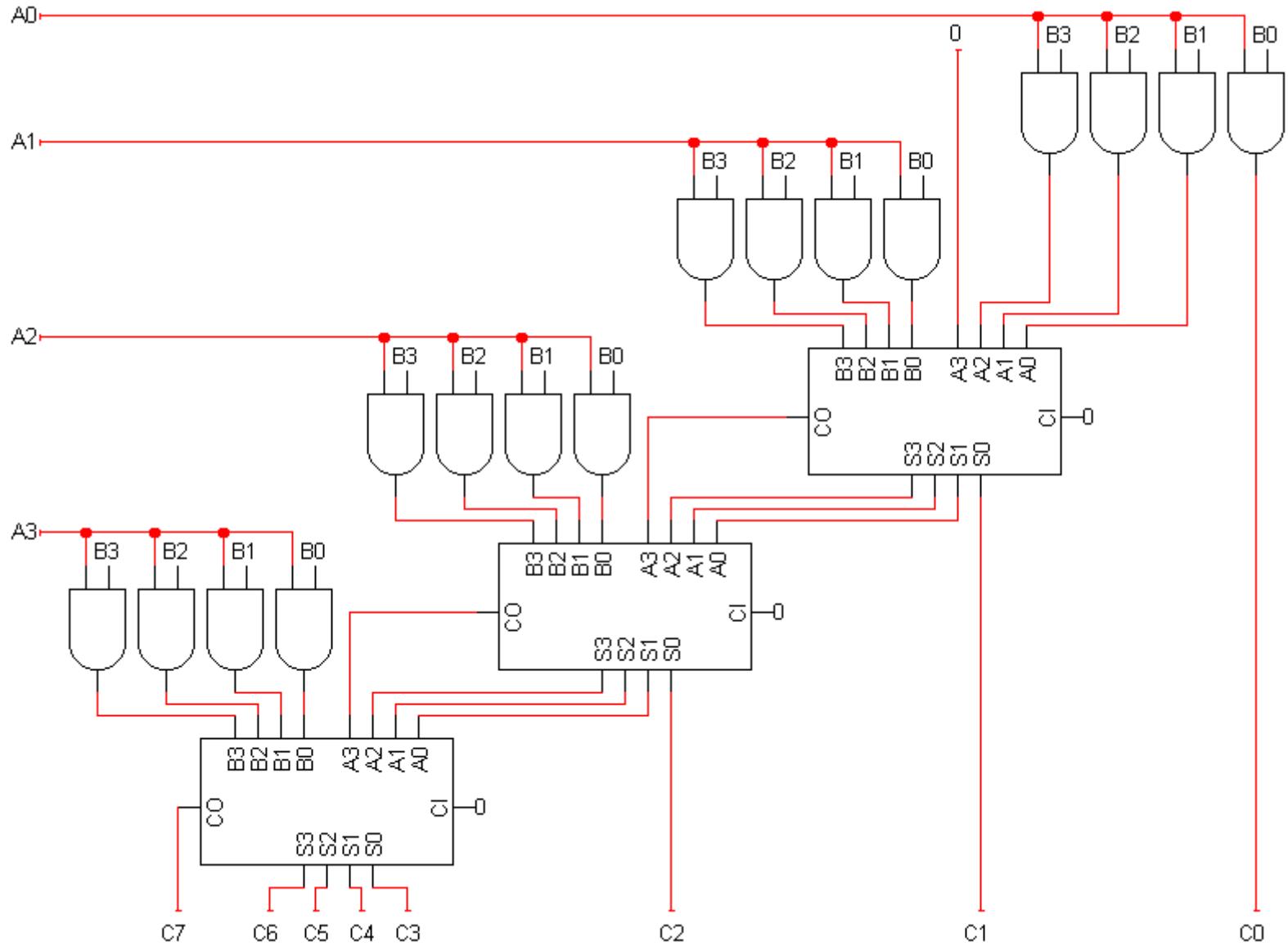
A 2x2 Binary Multiplier

- The AND gates produce the partial products
- For a 2-bit by 2-bit multiplier, we can just use two half adders to sum the partial products. In general, though, we'll need full adders
- Here C_3-C_0 are the product, not carries!

$$\begin{array}{r} & \text{B}_1 & \text{B}_0 \\ & \times & \\ & \text{A}_1 & \text{A}_0 \\ \hline & \text{A}_0\text{B}_1 & \text{A}_0\text{B}_0 \\ + & \text{A}_1\text{B}_1 & \text{A}_1\text{B}_0 \\ \hline & \text{C}_3 & \text{C}_2 & \text{C}_1 & \text{C}_0 \end{array}$$

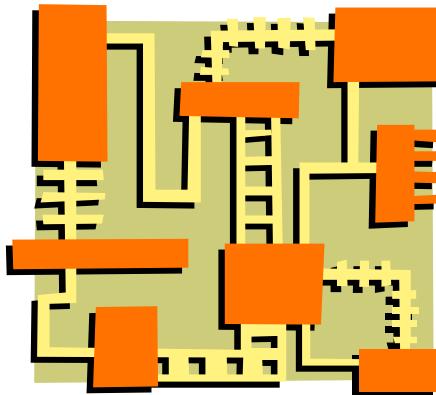


A 4x4 Multiplier Circuit



More on Multipliers

- Multipliers are very complex circuits
 - In general, when multiplying an m-bit number by an n-bit number:
 - There are n partial products, one for each bit of the multiplier
 - This requires n-1 adders, each of which can add m bits (the size of the multiplicand)
 - The circuit for 32-bit or 64-bit multiplication would be huge!



Multiplication: a special case

- In decimal, an easy way to multiply by 10 is to shift all the digits to the left, and tack a 0 to the right end

$$128 \times 10 = 1280$$

- We can do the same thing in binary. Shifting left is equivalent to multiplying by 2:

$$11 \times 10 = 110 \quad (\text{in decimal, } 3 \times 2 = 6)$$

- Shifting left twice is equivalent to multiplying by 4:

$$11 \times 100 = 1100 \quad (\text{in decimal, } 3 \times 4 = 12)$$

- As an aside, shifting to the right is equivalent to dividing by 2.

$$110 \div 10 = 11 \quad (\text{in decimal, } 6 \div 2 = 3)$$

Addition and Multiplication Summary

- Adder and multiplier circuits mimic human algorithms for addition and multiplication
- Adders and multipliers are built hierarchically
 - We start with half adders or full adders and work our way up
 - Building these functions from scratch with truth tables and K-maps would be pretty difficult
- The arithmetic circuits impose a limit on the number of bits that can be added. Exceeding this limit results in overflow
- There is a tradeoff between simple but slow circuits (ripple carry adders) and complex but fast circuits (carry lookahead adders)
- Multiplication and division by powers of 2 can be handled with simple shifting

Signed Numbers

- The arithmetic we did so far was limited to **unsigned** (positive) integers
- How about negative numbers and subtraction?
- We'll look at three different ways of representing **signed numbers**
- How can we decide which representation is better?
 - The best one should result in the simplest and fastest operations
- We're mostly concerned with two particular operations:
 - Negating a signed number, or converting x into $-x$
 - Adding two signed numbers, or computing $x + y$

Signed Magnitude Representation

- Humans use a **signed-magnitude** system: we add **+** or **-** in front of a magnitude to indicate the sign
- We could do this in binary as well, by adding an extra **sign bit** to the front of our numbers. By convention:
 - A **0** sign bit represents a positive number
 - A **1** sign bit represents a negative number
- Examples:

$1101_2 = 13_{10}$ (a 4-bit unsigned number)

0 1101 = $+13_{10}$ (a positive number in 5-bit signed magnitude)

1 1101 = -13_{10} (a negative number in 5-bit signed magnitude)

$0100_2 = 4_{10}$ (a 4-bit unsigned number)

0 0100 = $+4_{10}$ (a positive number in 5-bit signed magnitude)

1 0100 = -4_{10} (a negative number in 5-bit signed magnitude)

Signed Magnitude Operations

- Negating a signed-magnitude number is trivial: just change the sign bit from 0 to 1, or vice versa
- Adding numbers is difficult, though. Signed magnitude is basically what people use, so think about the grade-school approach to addition. It's based on comparing the signs of the augend and addend:
 - If they have the same sign, add the magnitudes and keep that sign
 - If they have different signs, then subtract the smaller magnitude from the larger one. The sign of the number with the larger magnitude is the sign of the result
- This method of subtraction would lead to a rather complex circuit.

$$\begin{array}{r} + 3 \quad 7 \quad 9 \\ + -6 \quad 4 \quad 7 \\ \hline -2 \quad 6 \quad 8 \end{array}$$

because

$$\begin{array}{r} 5 \quad 13 \quad 17 \\ 6 \quad 4 \quad 7 \\ - 3 \quad 7 \quad 9 \\ \hline 2 \quad 6 \quad 8 \end{array}$$

One's Complement Representation

- A different approach, **one's complement**, negates numbers by complementing each bit of the number
- We keep the sign bits: 0 for positive numbers, and 1 for negative. The sign bit is complemented along with the rest of the bits
- Examples:

$$1101_2 = 13_{10} \quad (\text{a 4-bit unsigned number})$$

$$\begin{matrix} 0 \\ 1101 \end{matrix} = +13_{10} \quad (\text{a positive number in 5-bit one's complement})$$

$$\begin{matrix} 1 \\ 0010 \end{matrix} = -13_{10} \quad (\text{a negative number in 5-bit one's complement})$$

$$0100_2 = 4_{10} \quad (\text{a 4-bit unsigned number})$$

$$\begin{matrix} 0 \\ 0100 \end{matrix} = +4_{10} \quad (\text{a positive number in 5-bit one's complement})$$

$$\begin{matrix} 1 \\ 1011 \end{matrix} = -4_{10} \quad (\text{a negative number in 5-bit one's complement})$$

Why is it called “one’s complement?”

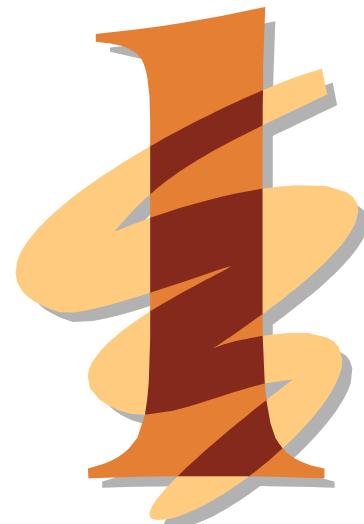
- Complementing a single bit is equivalent to subtracting it from 1

$$0' = 1, \text{ and } 1 - 0 = 1$$

$$1' = 0, \text{ and } 1 - 1 = 0$$

- Similarly, complementing each bit of an n-bit number is equivalent to subtracting that number from $2^n - 1$ (111...111)
- For example, we can negate the 5-bit number 01101
 - Here $n=5$, and $2^n - 1 = 31_{10} = 11111_2$
 - Subtracting 01101 from 11111 yields 10010

$$\begin{array}{r} 1 1 1 1 1 \\ - 0 1 1 0 1 \\ \hline 1 0 0 1 0 \end{array}$$



One's Complement Addition

- To add one's complement numbers:
 - First do unsigned addition on the numbers, *including* the sign bits
 - Then take the carry out and add it to the sum
- Two examples:

$$\begin{array}{r} 0111 \quad (+7) \\ + 1011 \quad + (-4) \\ \hline 1 \textcolor{blue}{0010} \end{array}$$

$$\begin{array}{r} \textcolor{blue}{0010} \\ + \textcolor{pink}{1} \\ \hline 0011 \quad (+3) \end{array}$$

$$\begin{array}{r} 0011 \quad (+3) \\ + 0010 \quad + (+2) \\ \hline 0 \textcolor{blue}{0101} \end{array}$$

$$\begin{array}{r} \textcolor{blue}{0101} \\ + \textcolor{pink}{0} \\ \hline 0101 \quad (+5) \end{array}$$

- This is simpler and more uniform than signed magnitude addition

Two's Complement

- Our final idea is **two's complement**. To negate a number, complement each bit (just as for ones' complement) and then add 1
- Examples:

$1101_2 = 13_{10}$ (a 4-bit unsigned number)

$0\ 1101 = +13_{10}$ (a positive number in 5-bit two's complement)

$1\ 0010 = -13_{10}$ (a negative number in 5-bit *ones'* complement)

$1\ 0011 = -13_{10}$ (a negative number in 5-bit two's complement)

$0100_2 = 4_{10}$ (a 4-bit unsigned number)

$0\ 0100 = +4_{10}$ (a positive number in 5-bit two's complement)

$1\ 1011 = -4_{10}$ (a negative number in 5-bit *ones'* complement)

$1\ 1100 = -4_{10}$ (a negative number in 5-bit two's complement)

More about two's complement

- Two other equivalent ways to negate two's complement numbers:

- You can subtract an n-bit two's complement number from 2^n

$$\begin{array}{r} 100000 \\ - 01101 \quad (+13_{10}) \\ \hline 10011 \quad (-13_{10}) \end{array}$$

$$\begin{array}{r} 100000 \\ - 00100 \quad (+4_{10}) \\ \hline 11100 \quad (-4_{10}) \end{array}$$

- You can complement all of the bits to the left of the rightmost 1

01101 = $+13_{10}$ (a positive number in two's complement)

10011 = -13_{10} (a negative number in two's complement)

00100 = $+4_{10}$ (a positive number in two's complement)

11100 = -4_{10} (a negative number in two's complement)

Two's Complement Addition

- Negating a two's complement number takes a bit of work, but addition is much easier than with the other two systems
- To find $A + B$, you just have to:
 - Do unsigned addition on A and B , including their sign bits
 - Ignore any carry out
- For example, to find $0111 + 1100$, or $(+7) + (-4)$:
 - First add $0111 + 1100$ as unsigned numbers:

$$\begin{array}{r} 0111 \\ + 1100 \\ \hline 10011 \end{array}$$

- Discard the carry out (1)
- The answer is $0011 (+3)$



Another two's complement example

- To further convince you that this works, let's try adding two negative numbers— $1101 + 1110$, or $(-3) + (-2)$ in decimal
- Adding the numbers gives 11011:

$$\begin{array}{r} 1101 \\ + 1110 \\ \hline 11011 \end{array}$$

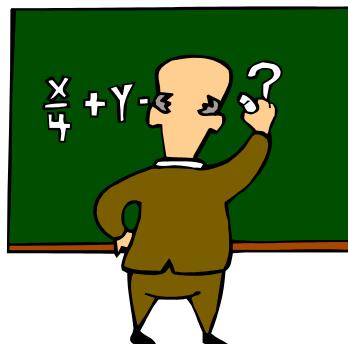
- Dropping the carry out (1) leaves us with the answer, 1011 (-5).

Why does this work?

- For n-bit numbers, the negation of B in two's complement is $2^n - B$ (this is one of the alternative ways of negating a two's-complement number)

$$\begin{aligned}A - B &= A + (-B) \\&= A + (2^n - B) \\&= (A - B) + 2^n\end{aligned}$$

- If $A \geq B$, then $(A - B)$ is a positive number, and 2^n represents a carry out of 1. Discarding this carry out is equivalent to subtracting 2^n , which leaves us with the desired result $(A - B)$
- If $A < B$, then $(A - B)$ is a negative number and we have $2^n - (B - A)$. This corresponds to the desired result, $-(A - B)$, in two's complement form.



Comparing the signed number systems

- Positive numbers are the same in all three representations
- Signed magnitude and one's complement have two ways of representing 0. This makes things more complicated
- Two's complement has asymmetric ranges; there is one more negative number than positive number. Here, you can represent -8 but not +8
- However, two's complement is preferred because it has only one 0, and its addition algorithm is the simplest

Decimal	S.M.	1's comp.	2's comp.
7	0111	0111	0111
6	0110	0110	0110
5	0101	0101	0101
4	0100	0100	0100
3	0011	0011	0011
2	0010	0010	0010
1	0001	0001	0001
0	0000	0000	0000
-0	1000	1111	-
-1	1001	1110	1111
-2	1010	1101	1110
-3	1011	1100	1101
-4	1100	1011	1100
-5	1101	1010	1011
-6	1110	1001	1010
-7	1111	1000	1001
-8	-	-	1000

Ranges of the signed number systems

- How many negative and positive numbers can be represented in each of the different systems on the previous page?

	Unsigned	Signed Magnitude	One's complement	Two's complement
Smallest	0000 (0)	1111 (-7)	1000 (-7)	1000 (-8)
Largest	1111 (15)	0111 (+7)	0111 (+7)	0111 (+7)

- In general, with n-bit numbers including the sign, the ranges are:

	Unsigned	Signed Magnitude	One's complement	Two's complement
Smallest	0	$-(2^{n-1}-1)$	$-(2^{n-1}-1)$	-2^{n-1}
Largest	$2^n - 1$	$+(2^{n-1}-1)$	$+(2^{n-1}-1)$	$+(2^{n-1}-1)$

Converting signed numbers to decimal

- Convert 110101 to decimal, assuming this is a number in:

(a) signed magnitude format

(b) ones' complement

(c) two's complement

Example solution

- Convert 110101 to decimal, assuming this is a number in:

Since the sign bit is 1, this is a negative number. The easiest way to find the magnitude is to convert it to a positive number.

- (a) signed magnitude format

Negating the original number, 110101, gives 010101, which is +21 in decimal. So 110101 must represent -21.

- (b) ones' complement

Negating 110101 in ones' complement yields 001010 = $+10_{10}$, so the original number must have been -10_{10} .

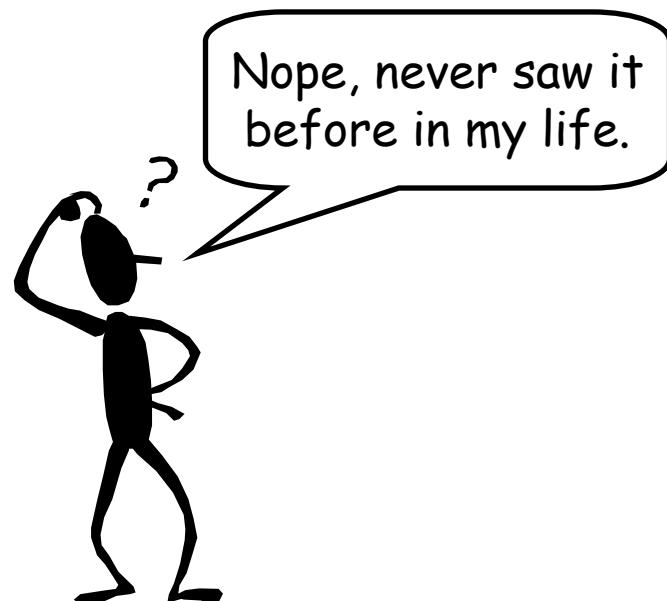
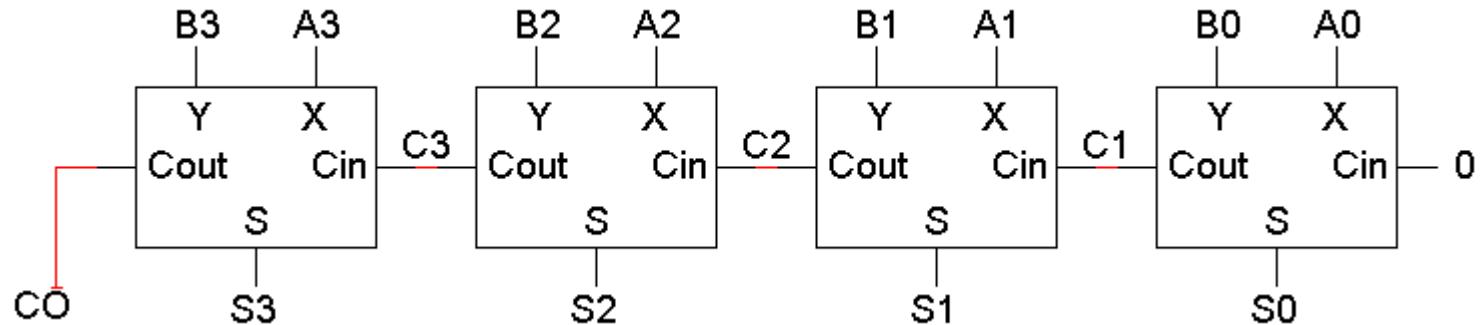
- (c) two's complement

Negating 110101 in two's complement gives 001011 = 11_{10} , which means $110101 = -11_{10}$.

- The most important point here is that a binary number has *different* meanings depending on which representation is assumed

Our four-bit unsigned adder circuit

- Here is the four-bit unsigned addition circuit



Making a subtraction circuit

- We could build a subtraction circuit directly, similar to the way we made unsigned adders
- However, by using two's complement we can convert any subtraction problem into an addition problem. Algebraically,

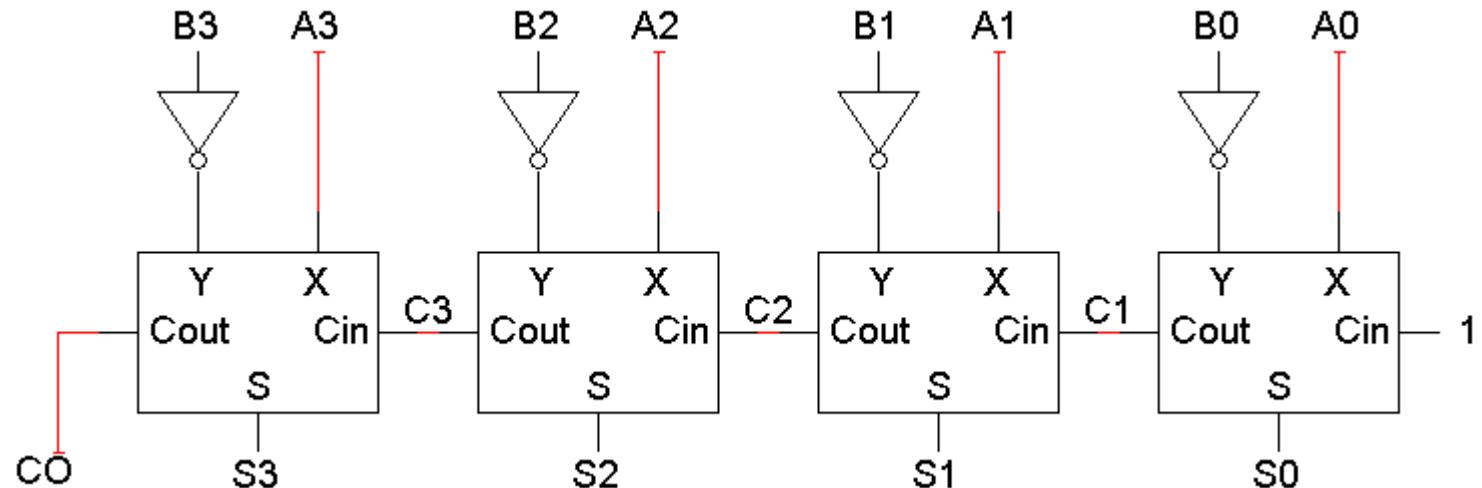
$$A - B = A + (-B)$$

- So to subtract B from A, we can instead *add* the negation of B to A
- This way we can re-use the unsigned adder hardware



A two's complement subtraction circuit

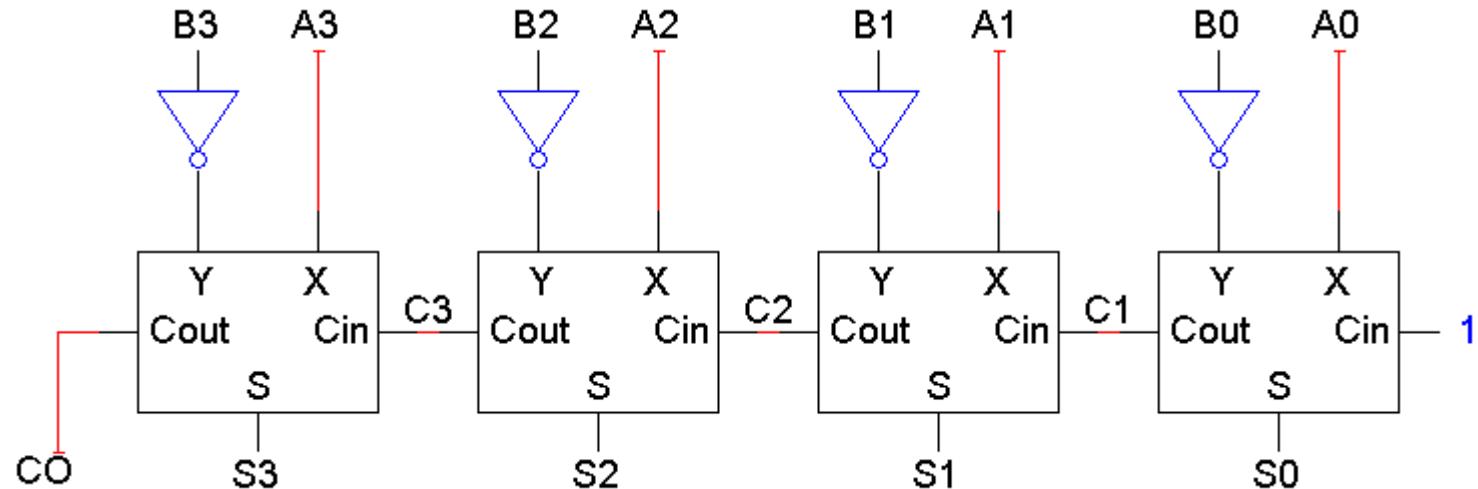
- To find $A - B$ with an adder, we'll need to:
 - Complement each bit of B
 - Set the adder's carry in to 1
- The net result is $A + B' + 1$, where $B' + 1$ is the two's complement negation of B



- Remember that A_3, B_3 and S_3 here are actually sign bits

Small differences

- The only differences between the adder and subtractor circuits are:
 - The subtractor has to negate B3 B2 B1 B0
 - The subtractor sets the initial carry in to 1, instead of 0



- Not too hard to make one circuit that does *both* addition and subtraction

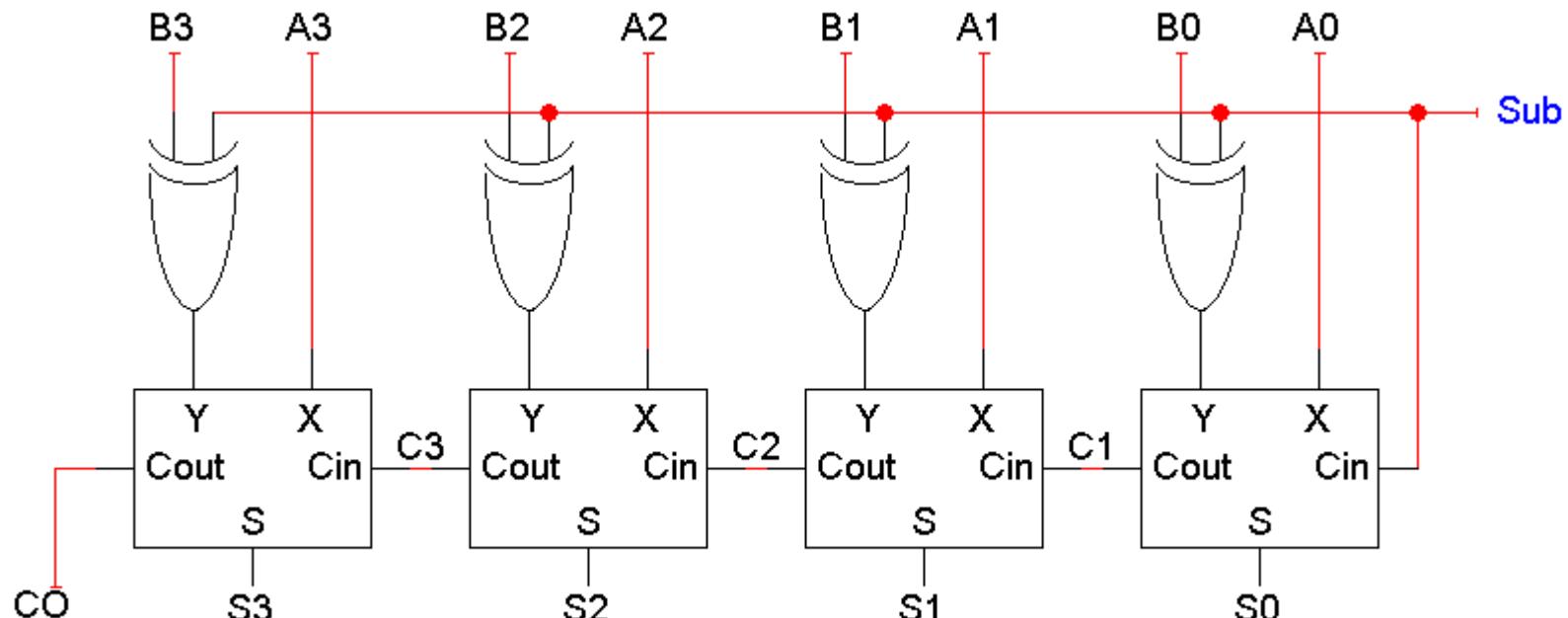
An Adder-Subracter Circuit

- XOR gates let us selectively complement the B input

$$X \oplus 0 = X$$

$$X \oplus 1 = X'$$

- When **Sub = 0**, the XOR gates output B3' B2' B1' B0' and the carry in is 0. The adder output will be A + B + 0, or just A + B
- When **Sub = 1**, the XOR gates output B3 B2 B1 B0 and the carry in is 1. Thus, the adder output will be a two's complement subtraction, A - B



Signed Overflow

- With two's complement and a 4-bit adder, for example, the largest representable decimal number is +7, and the smallest is -8
- What if you try to compute $4 + 5$, or $(-4) + (-5)$?

$$\begin{array}{r} 0100 \quad (+4) \\ + 0101 \quad (+5) \\ \hline 01001 \quad (-7) \end{array} \qquad \begin{array}{r} 1100 \quad (-4) \\ + 1011 \quad (-5) \\ \hline 10111 \quad (+7) \end{array}$$

- We cannot just include the carry out to produce a five-digit result, as for unsigned addition. If we did, $(-4) + (-5)$ would result in +23!
- Also, unlike the case with unsigned numbers, the carry out *cannot* be used to detect overflow
 - In the example on the left, the carry out is 0 but there *is* overflow
 - Conversely, there are situations where the carry out is 1 but there is *no* overflow

Detecting signed overflow

- The easiest way to detect signed overflow is to look at all the sign bits

$$\begin{array}{r} 0100 \quad (+4) \\ + 0101 \quad (+5) \\ \hline 01001 \quad (-7) \end{array}$$

$$\begin{array}{r} 1100 \quad (-4) \\ + 1011 \quad (-5) \\ \hline 10111 \quad (+7) \end{array}$$

- Overflow occurs only in the two situations above:
 - If you add two *positive* numbers and get a *negative* result
 - If you add two *negative* numbers and get a *positive* result
- Overflow cannot occur if you add a positive number to a negative number.
Do you see why?

Sign Extension

- In everyday life, decimal numbers are assumed to have an infinite number of 0s in front of them. This helps in “lining up” numbers
- To subtract 231 and 3, for instance, you can imagine:

$$\begin{array}{r} 231 \\ - 003 \\ \hline 228 \end{array}$$

- You need to be careful in extending signed binary numbers, because the leftmost bit is the *sign* and not part of the magnitude
- If you just add 0s in front, you might accidentally change a negative number into a positive one!
- For example, going from 4-bit to 8-bit numbers:
 - 0101 (+5) should become 0000 0101 (+5)
 - But 1100 (-4) should become 1111 1100 (-4)
- The proper way to extend a signed binary number is to replicate the sign bit, so the sign is preserved

Subtraction Summary

- A good representation for negative numbers makes subtraction hardware much easier to design
 - Two's complement is used most often (although signed magnitude shows up sometimes, such as in floating-point systems)
 - Using two's complement, we can build a subtractor with minor changes to the adder
 - We can also make a single circuit which can both add and subtract
- Overflow is still a problem, but signed overflow is very different from the unsigned overflow
- Sign extension is needed to properly "lengthen" negative numbers

Flip-Flops

- Last time, we saw how latches can be used as memory in a circuit
- Latches introduce new problems:
 - We need to know when to enable a latch
 - We also need to quickly disable a latch
 - In other words, it's difficult to control the timing of latches in a large circuit
- We solve these problems with two new elements: clocks and flip-flops
 - Clocks tell us when to write to our memory
 - Flip-flops allow us to quickly write the memory at clearly defined times
 - Used together, we can create circuits without worrying about the memory timing

Using latches in real life

- We can connect some latches, acting as memory, to an ALU



- Let's say these latches contain some value that we want to increment
 - The ALU should read the current latch value
 - It applies the " $G = X + 1$ " operation
 - The incremented value is stored back into the latches
- At this point, we have to stop the cycle, so the latch value doesn't get incremented again by accident
- One convenient way to break the loop is to disable the latches

Making latches work right

- Our example used latches as memory for an ALU
 - Let's say there are four latches initially storing 0000
 - We want to use an ALU to increment that value to 0001
- Normally the latches should be disabled, to prevent unwanted data from being accidentally stored
 - In our example, the ALU can read the current latch contents, 0000, and compute their increment, 0001
 - But the new value cannot be stored back while the latch is disabled

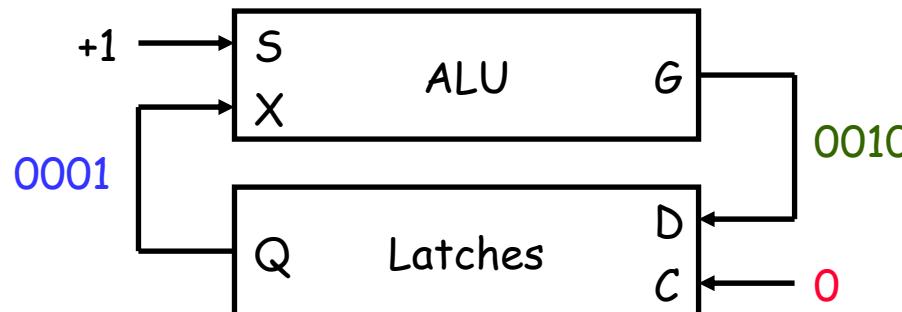


Writing to the latches

- After the ALU has finished its increment operation, the latch can be enabled, and the updated value is stored.



- The latch must be quickly disabled again, *before* the ALU has a chance to read the new value 0001 and produce a new result 0010.



Two main issues

- So to use latches correctly within a circuit, we have to:
 - Keep the latches disabled until new values are ready to be stored
 - Enable the latches just long enough for the update to occur
- There are two main issues we need to address:
 - ▶ How do we know exactly when the new values are ready?

We'll add another signal to our circuit. When this new signal becomes 1, the latches will know that the ALU computation has completed and data is ready to be stored
 - ▶ How can we enable and then quickly disable the latches?

This can be done by combining latches together in a special way, to form what are called flip-flops

Clocks and synchronization

- A **clock** is a special device whose output continuously alternates between 0 and 1.

clock period



- The time it takes the clock to change from 1 to 0 and back to 1 is called the **clock period**, or **clock cycle time**
- The **clock frequency** is the inverse of the clock period. The unit of measurement for frequency is the **hertz**
- Clocks are often used to synchronize circuits
 - They generate a repeating, predictable pattern of 0s and 1s that can trigger certain events in a circuit, such as writing to a latch
 - If several circuits share a common clock signal, they can coordinate their actions with respect to one another
- This is similar to how humans use real clocks for synchronization.

More about clocks

- Clocks are used extensively in computer architecture
- All processors run with an internal clock
 - Modern chips run at frequencies up to 3.2 GHz.
 - This works out to a cycle time as little as 0.31 ns!
- Memory modules are often rated by their clock speeds too—examples include "PC133" and "DDR400" memory
- Be careful...higher frequencies do not always mean faster machines!
 - You also have to consider how much work is actually being done during each clock cycle
 - How much stuff can really get done in just 0.31 ns?



Synchronizing our example

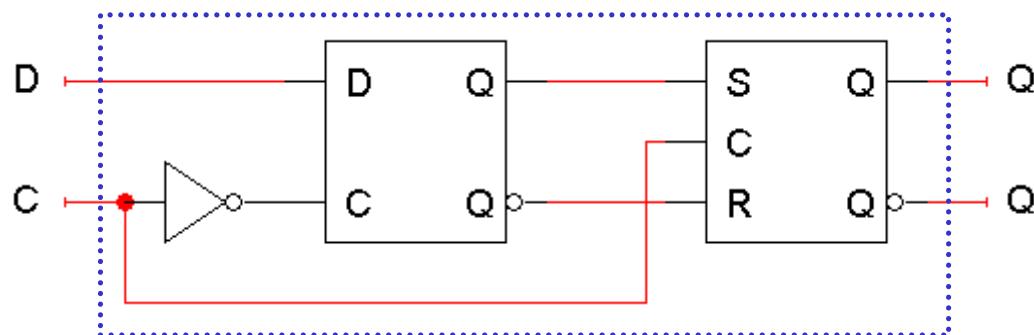
- We can use a clock to synchronize our latches with the ALU
 - The clock signal is connected to the latch control input C
 - The clock controls the latches. When it becomes 1, the latches will be enabled for writing



- The clock period must be set appropriately for the ALU
 - It should not be too short. Otherwise, the latches will start writing before the ALU operation has finished
 - The faster the ALU runs, the shorter the clock period can be
 - But in the current design, if the clock period is too large, it's a problem too..

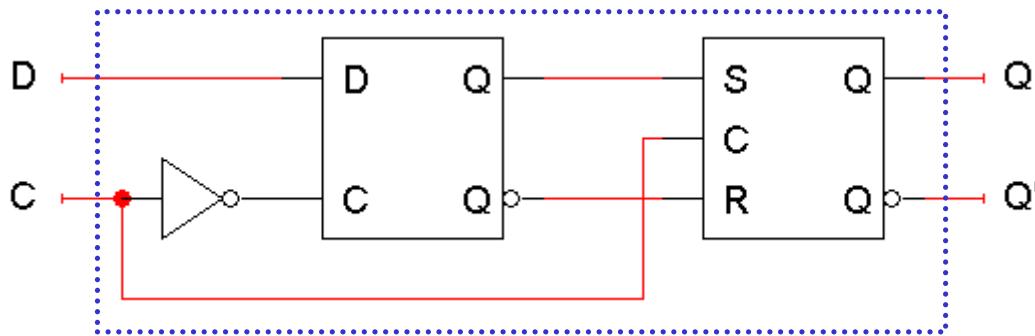
Flip-flops

- The second issue was how to enable a latch for just an instant
- Here is the internal structure of a **D flip-flop**
 - The flip-flop inputs are C and D , and the outputs are Q and Q'
 - The D latch on the left is the **master**, while the SR latch on the right is called the **slave**



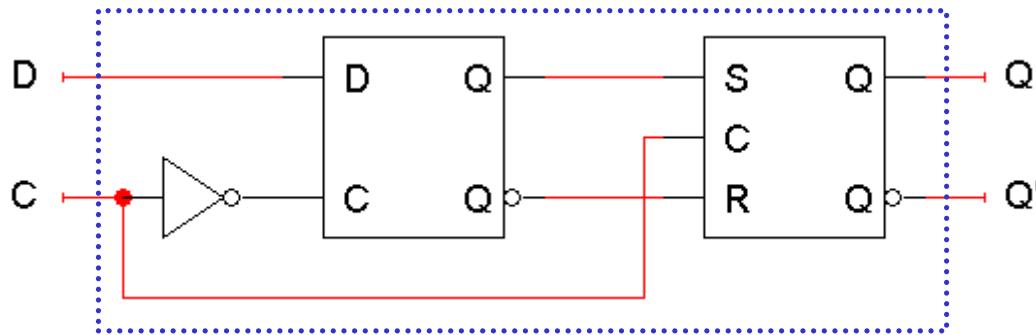
- Note the layout here
 - The flip-flop input D is connected directly to the master latch
 - The master latch output goes to the slave
 - The flip-flop outputs come directly from the slave latch

D flip-flops when $C=0$



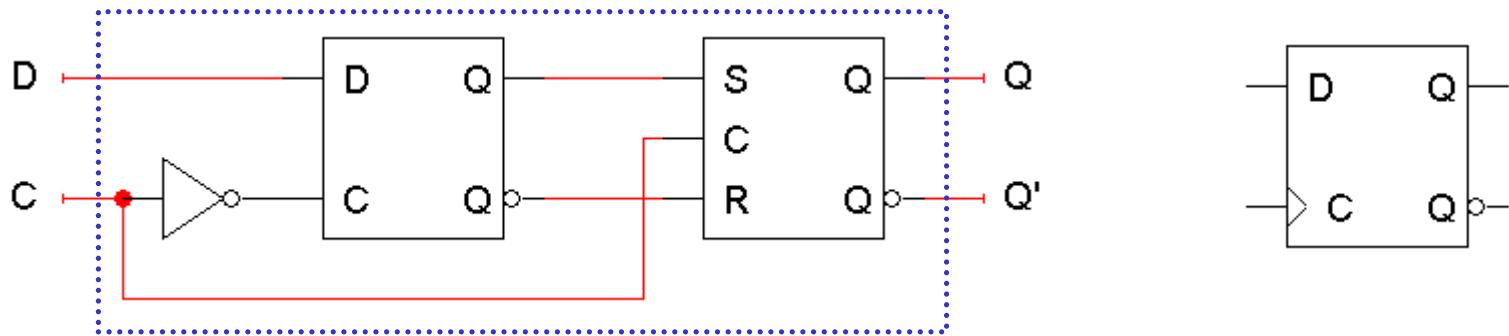
- The D flip-flop's control input C enables *either* the D latch or the SR latch, but not both
- When $C = 0$:
 - The master latch is enabled, and it monitors the flip-flop input D . Whenever D changes, the master's output changes too
 - The slave is disabled, so the D latch output has no effect on it. Thus, the slave just maintains the flip-flop's current state

D flip-flops when C=1



- As soon as C becomes 1, (i.e. on the rising edge of the clock)
 - The master is disabled. Its output will be the *last* D input value seen just before C became 1
 - Any subsequent changes to the D input while $C = 1$ have no effect on the master latch, which is now disabled
 - The slave latch is enabled. Its state changes to reflect the master's output, which again is the D input value from right when C became 1

Positive edge triggering

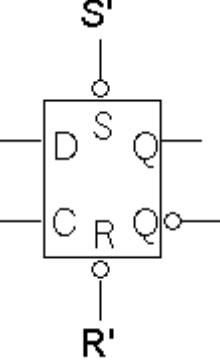


- This is called a **positive edge-triggered** flip-flop
 - The flip-flop output Q changes *only* after the positive edge of C
 - The change is based on the flip-flop input values that were present right at the positive edge of the clock signal
- The D flip-flop's behavior is similar to that of a D latch except for the positive edge-triggered nature, which is not explicit in this table

C	D	Q
0	x	No change
1	0	0 (reset)
1	1	1 (set)

Direct inputs

- One last thing to worry about... what is the starting value of Q?
- We could set the initial value synchronously, at the next positive clock edge, but this actually makes circuit design more difficult
- Instead, most flip-flops provide **direct**, or asynchronous, inputs that let you immediately set or clear the state
 - You would "reset" the circuit once, to initialize the flip-flops
 - The circuit would then begin its regular, synchronous operation



S'	R'	C	D	Q
0	0	x	x	Avoid!
0	1	x	x	1 (set)
1	0	x	x	0 (reset)
1	1	0	x	No change
1	1	1	0	0 (reset)
1	1	1	1	1 (set)

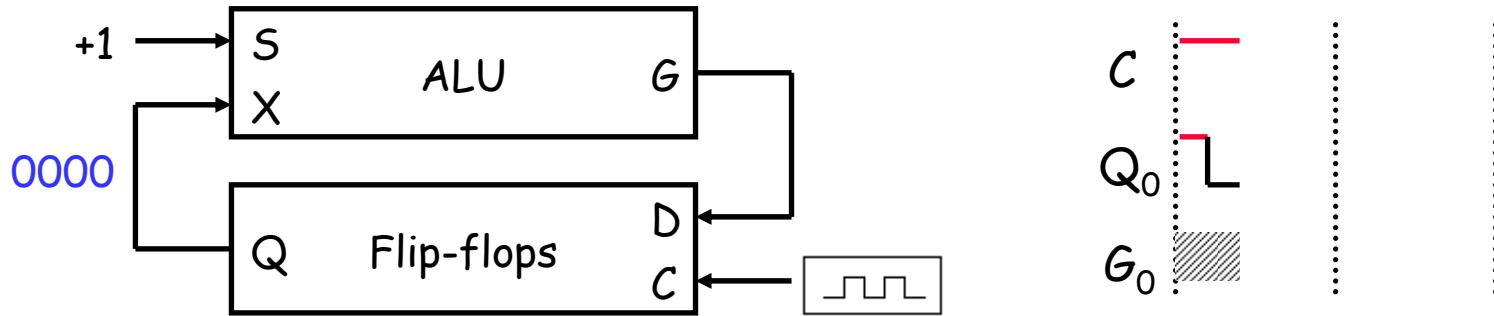
}

Direct inputs to set or
reset the flip-flop

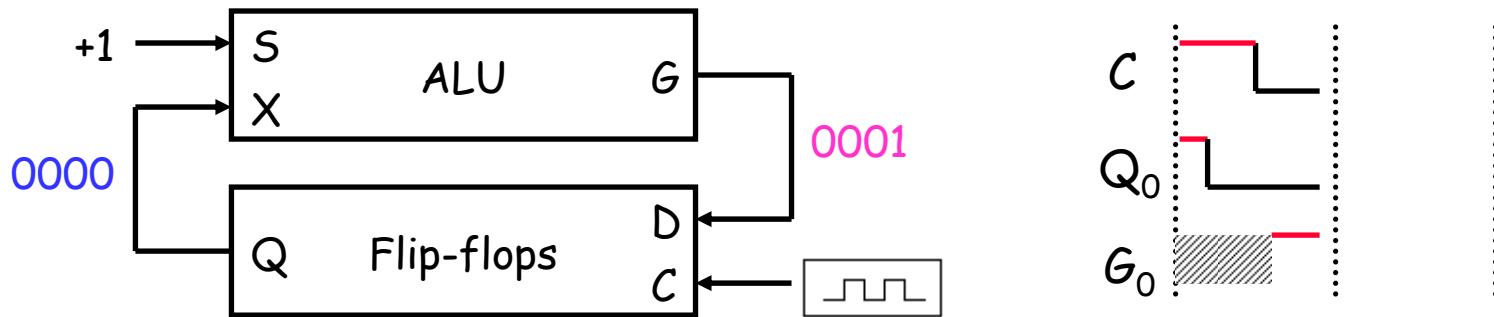
} $S'R' = 11$ for "normal"
operation of the D
flip-flop

Our example with flip-flops

- We can use the flip-flops' direct inputs to initialize them to 0000

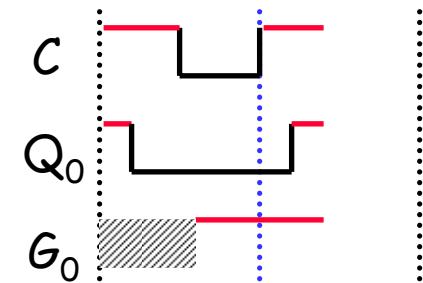
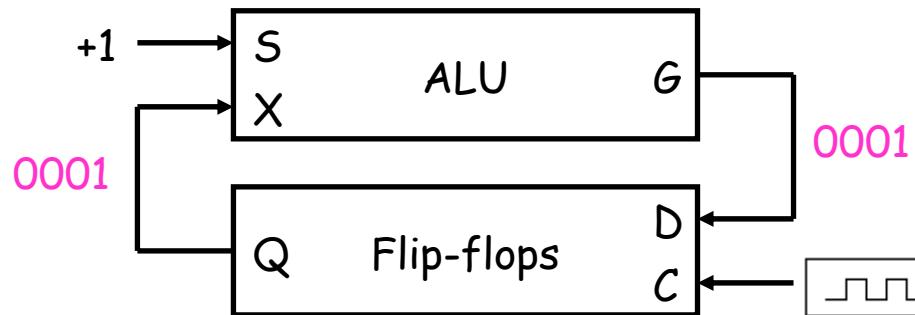


- During the clock cycle, the ALU outputs 0001, but this does not affect the flip-flops yet

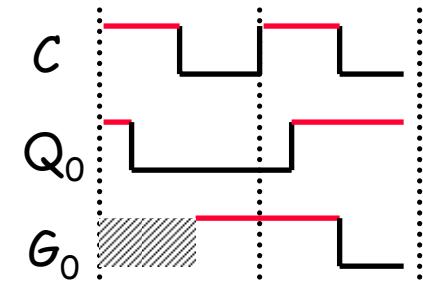
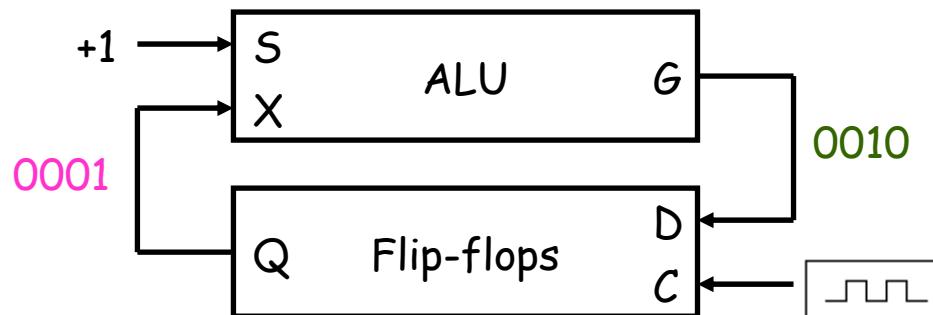


Example continued

- The ALU output is copied into the flip-flops at the next positive edge of the clock signal.

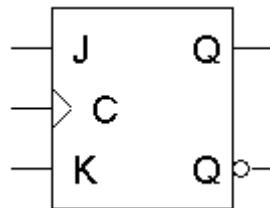


- The flip-flops automatically "shut off," and no new data can be written until the next positive clock edge... even though the ALU produces a new output.



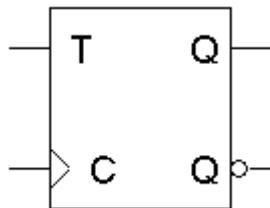
Flip-flop variations

- We can make different versions of flip-flops based on the D flip-flop, just like we made different latches based on the S'R' latch
- A **JK flip-flop** has inputs that act like S and R, but the inputs JK=11 are used to *complement* the flip-flop's current state



C	J	K	Q _{next}
0	x	x	No change
1	0	0	No change
1	0	1	0 (reset)
1	1	0	1 (set)
1	1	1	Q' _{current}

- A **T flip-flop** can only maintain or complement its current state.



C	T	Q _{next}
0	x	No change
1	0	No change
1	1	Q' _{current}

Characteristic tables

- The tables that we've made so far are called **characteristic tables**
 - They show the next state $Q(t+1)$ in terms of the current state $Q(t)$ and the inputs
 - For simplicity, the control input C is not usually listed.
 - Again, these tables don't indicate the positive edge-triggered behavior of the flip-flops that we'll be using.

D	$Q(t+1)$	Operation
0	0	Reset
1	1	Set

J	K	$Q(t+1)$	Operation
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q'(t)$	Complement

T	$Q(t+1)$	Operation
0	$Q(t)$	No change
1	$Q'(t)$	Complement

Characteristic equations

- We can also write **characteristic equations**, where the next state $Q(t+1)$ is defined in terms of the current state $Q(t)$ and inputs

D	$Q(t+1)$	Operation
0	0	Reset
1	1	Set

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

J	K	$Q(t+1)$	Operation
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q'(t)$	Complement

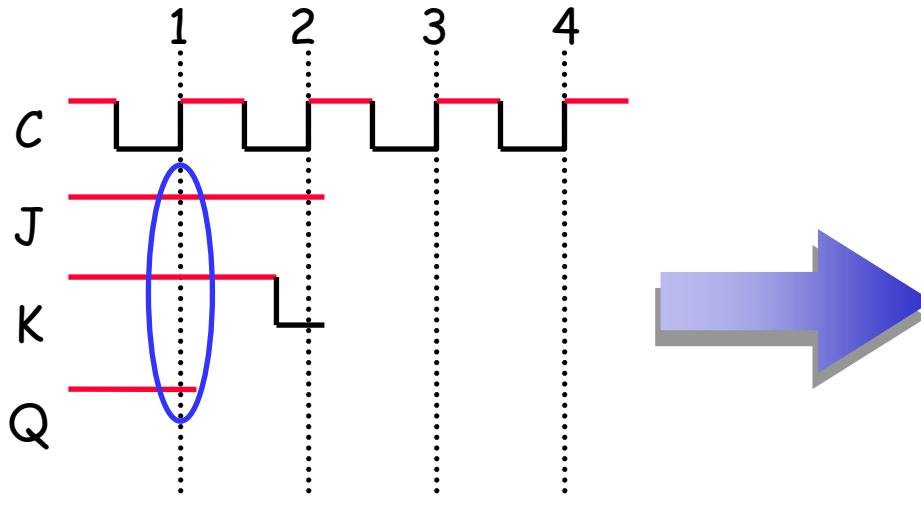
$$Q(t+1) = K'Q(t) + JQ'(t)$$

T	$Q(t+1)$	Operation
0	$Q(t)$	No change
1	$Q'(t)$	Complement

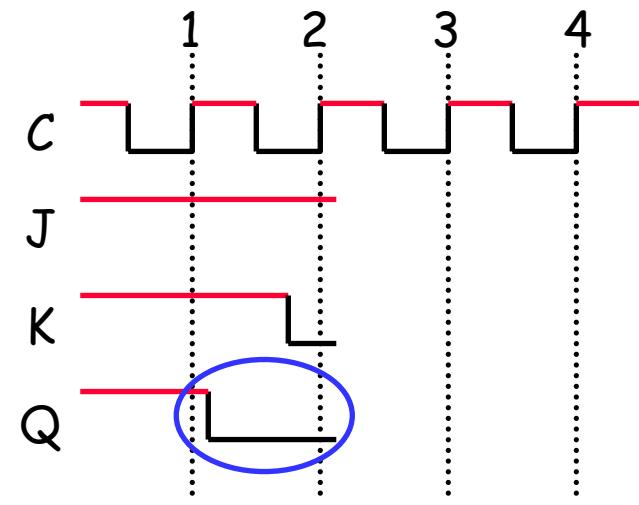
$$\begin{aligned} Q(t+1) &= T'Q(t) + TQ'(t) \\ &= T \oplus Q(t) \end{aligned}$$

Flip flop timing diagrams

- "Present state" and "next state" are relative terms
- In the example JK flip-flop timing diagram on the left, you can see that at the first positive clock edge, $J=1$, $K=1$ and $Q(1) = 1$
- We can use this information to find the "next" state, $Q(2) = Q(1)'$
- $Q(2)$ appears right after the first positive clock edge, as shown on the right. It will not change again until after the second clock edge



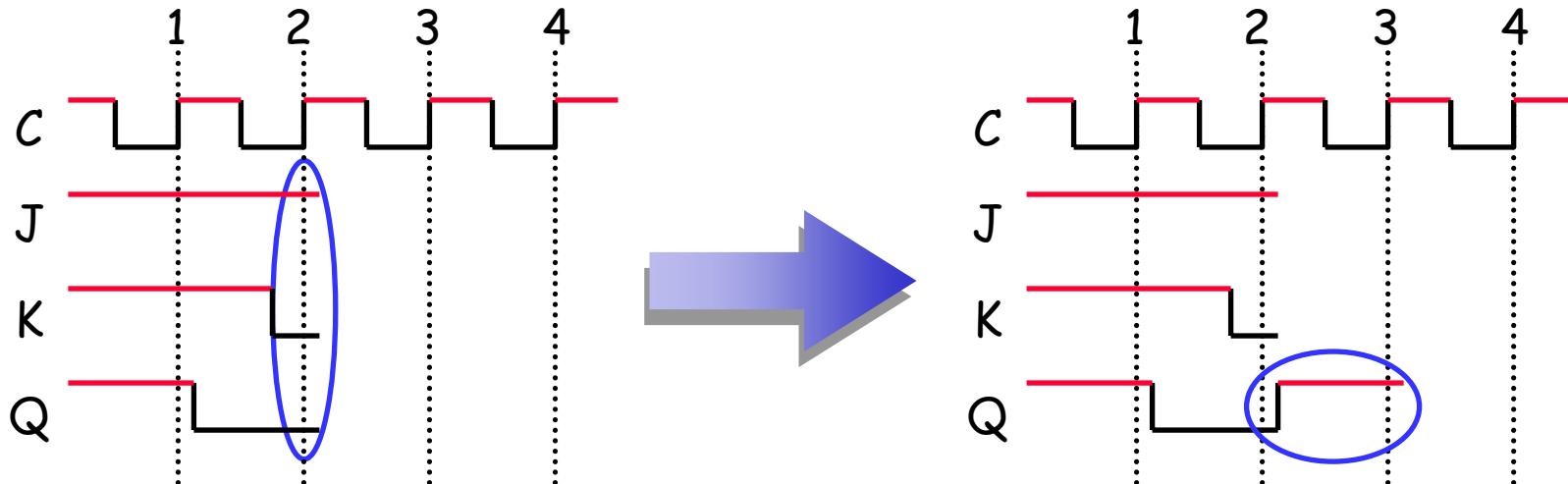
These values at clock cycle 1...



... determine the "next" Q

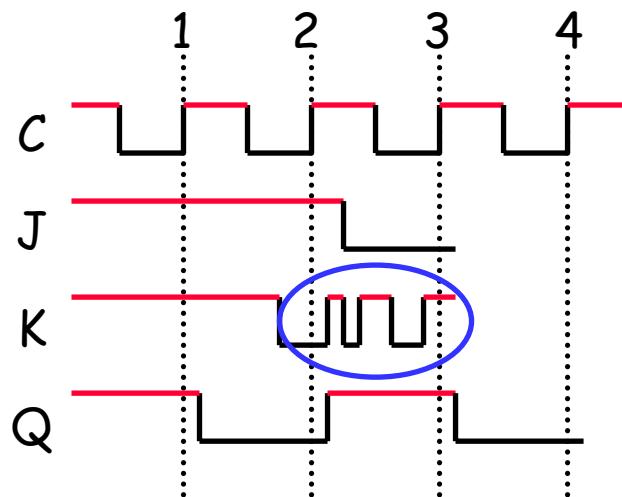
"Present" and "next" are relative

- Similarly, the values of J, K and Q at the second positive clock edge can be used to find the value of Q during the third clock cycle
- When we do this, $Q(2)$ is now referred to as the "present" state, and $Q(3)$ is now the "next" state



Positive edge triggered

- One final point to repeat: the flip-flop outputs are affected only by the input values *at the positive edge*
 - In the diagram below, K changes rapidly between the second and third positive edges
 - But it's only the input values at the third clock edge ($K=1$, and $J=0$ and $Q=1$) that affect the next state, so here Q changes to 0
- This is a fairly simple timing model. In real life there are "setup times" and "hold times" to worry about as well, to account for internal and external delays



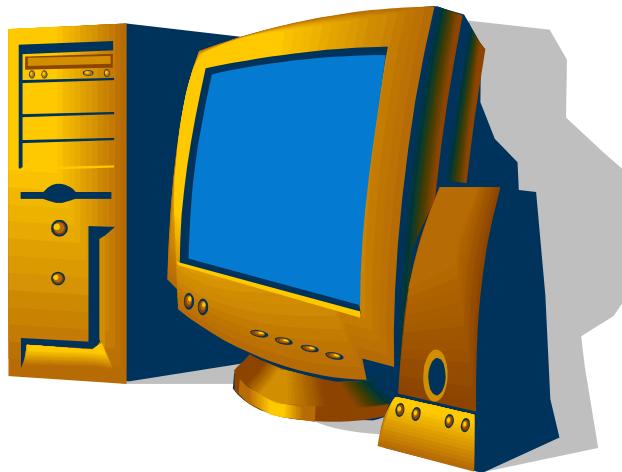
Summary

- To use memory in a larger circuit, we need to:
 - Keep the latches disabled until new values are ready to be stored
 - Enable the latches just long enough for the update to occur
- A clock signal is used to synchronize circuits. The cycle time reflects how long combinational operations take
- Flip-flops further restrict the memory writing interval, to just the positive edge of the clock signal
 - This ensures that memory is updated only once per clock cycle
 - There are several different kinds of flip-flops, but they all serve the same basic purpose of storing bits
- Next week we'll talk about how to analyze and design sequential circuits that use flip-flops as memory

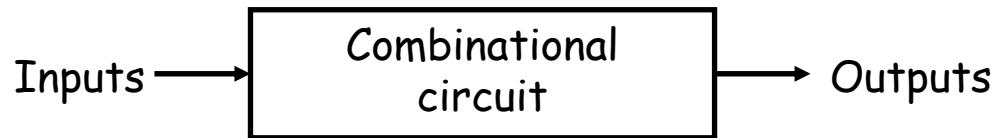
-
- Chapter 5: Sequential Circuits (LATCHES)

Latches

- We focuses on **sequential circuits**, where we add memory to the hardware that we've already seen
- Our schedule will be very similar to before:
 - We first show how primitive memory units are built
 - Then we talk about analysis and design of sequential circuits
 - We also see common sequential devices like registers and counters

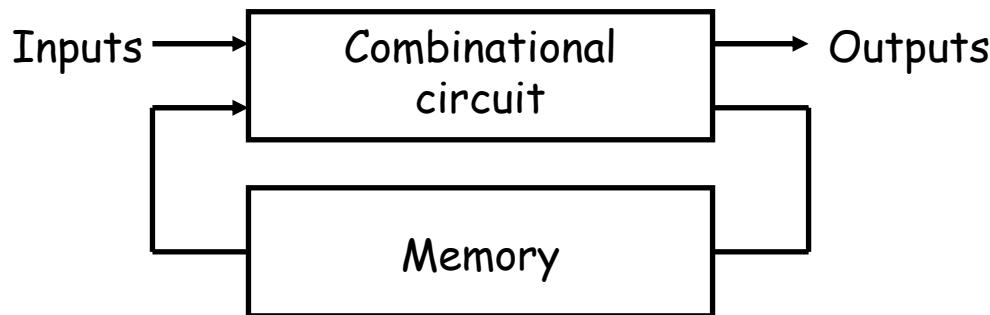


Combinational circuits



- So far we've just worked with **combinational circuits**, where applying the same inputs always produces the same outputs
- This corresponds to a mathematical function, where every input has a single, unique output

Sequential circuits



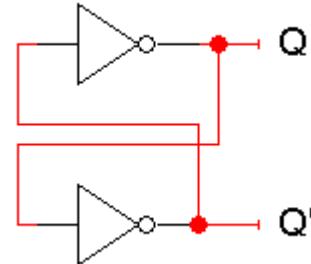
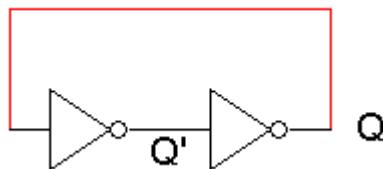
- In contrast, the outputs of a **sequential circuit** depend on not only the inputs, but also the **state**, or the current contents of some memory
- This makes things more difficult to understand, since the same inputs can yield *different* outputs, depending on what's stored in memory
- The memory contents can also change as the circuit runs
- We'll need some new techniques for analyzing and designing sequential circuits.

What exactly is memory?

- A memory should have at least three properties
 1. It should be able to hold a value.
 2. You should be able to *read* the value that was saved
 3. You should be able to *change* the value that's saved
- We'll start with the simplest case, a one-bit memory
 1. It should be able to hold a single bit, 0 or 1
 2. You should be able to read the bit that was saved
 3. You should be able to change the value. Since there's only a single bit, there are only two choices:
 - **Set** the bit to 1
 - **Reset**, or **clear**, the bit to 0

The basic idea of storage

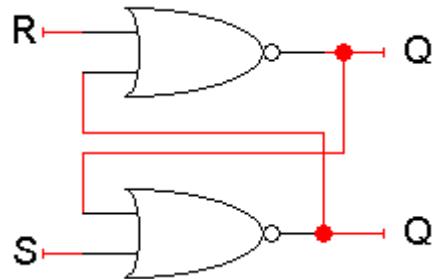
- How can a circuit “remember” anything, when it’s just a bunch of gates that produce outputs according to the inputs?
- The basic idea is to make a loop, so the circuit outputs are also inputs
- Here is one initial attempt, shown with two equivalent layouts:



- Does this satisfy the properties of memory?
 - These circuits “remember” Q , because its value never changes (Similarly, Q' never changes either.)
 - We can also “read” Q , by attaching a probe or another circuit
 - But we can’t *change* Q ! There are no external inputs here, so we can’t control whether $Q=1$ or $Q=0$.

A really confusing circuit

- Let's use NOR gates instead of inverters. The **SR latch** below has two inputs S and R, which will let us control the outputs Q and Q'



- Here Q and Q' feed back into the circuit. They're not only outputs, they're also inputs!
- To figure out how Q and Q' change, we have to look at not only the inputs S and R, but also the *current* values of Q and Q':

$$Q_{\text{next}} = (R + Q'_{\text{current}})'$$

$$Q'_{\text{next}} = (S + Q_{\text{current}})'$$

- Let's see how different input values for S and R affect this thing

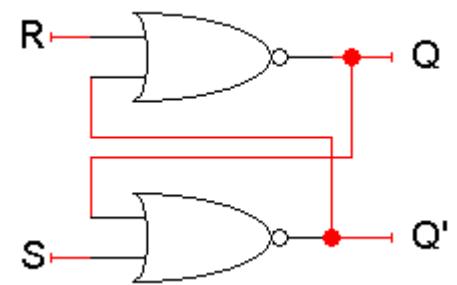
Storing a value: SR = 00

- What if S = 0 and R = 0?
- The equations on the right reduce to:

$$Q_{\text{next}} = (0 + Q'_{\text{current}})' = Q_{\text{current}}$$

$$Q'_{\text{next}} = (0 + Q_{\text{current}})' = Q'_{\text{current}}$$

- So when SR = 00, then $Q_{\text{next}} = Q_{\text{current}}$. Whatever value Q has, it keeps
- This is exactly what we need to **store** values in the latch.

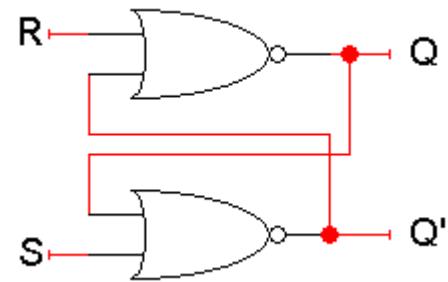


$$Q_{\text{next}} = (R + Q'_{\text{current}})'$$

$$Q'_{\text{next}} = (S + Q_{\text{current}})'$$

Setting the latch: $SR = 10$

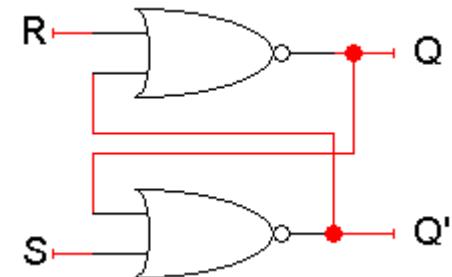
- What if $S = 1$ and $R = 0$?
- Since $S = 1$, Q'_{next} is 0, *regardless* of Q_{current} :
$$Q'_{\text{next}} = (1 + Q_{\text{current}})' = 0$$
- Then, this new value of Q' goes into the top NOR gate, along with $R = 0$.
$$Q_{\text{next}} = (0 + 0)' = 1$$
- So when $SR = 10$, then $Q'_{\text{next}} = 0$ and $Q_{\text{next}} = 1$
- This is how you **set** the latch to 1. The S input stands for "set"
- Notice that it can take up to two steps (two gate delays) from the time S becomes 1 to the time Q_{next} becomes 1
- But once Q_{next} becomes 1, the outputs will stop changing. This is a **stable state**



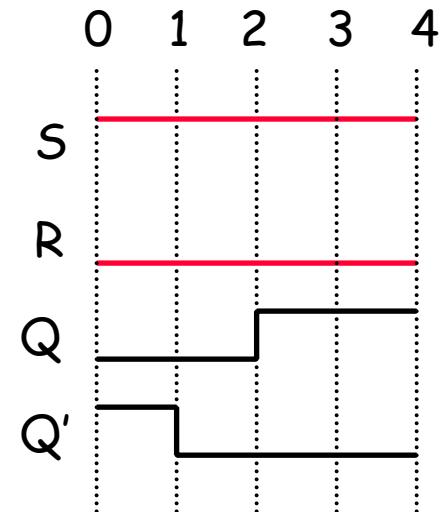
$$Q_{\text{next}} = (R + Q'_{\text{current}})'$$
$$Q'_{\text{next}} = (S + Q_{\text{current}})'$$

Latch delays

- Timing diagrams are especially useful in understanding how sequential circuits work
- Here is a diagram which shows an example of how our latch outputs change with inputs SR=10
 - 0. Suppose that initially, $Q = 0$ and $Q' = 1$
 - 1. Since $S=1$, Q' will change from 1 to 0 after one NOR-gate delay (marked by vertical lines in the diagram for clarity)
 - 2. This change in Q' , along with $R=0$, causes Q to become 1 after another gate delay
 - 3. The latch then stabilizes until S or R change again



$$Q_{\text{next}} = (R + Q'_{\text{current}})'$$
$$Q'_{\text{next}} = (S + Q_{\text{current}})'$$

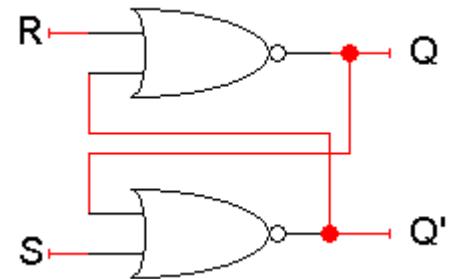


Resetting the latch: SR = 01

- What if $S = 0$ and $R = 1$?
- Since $R = 1$, Q_{next} is 0, regardless of Q_{current} :
$$Q_{\text{next}} = (1 + Q'_{\text{current}})' = 0$$
- Then, this new value of Q goes into the bottom NOR gate, where $S = 0$

$$Q'_{\text{next}} = (0 + 0)' = 1$$

- So when $SR = 01$, then $Q_{\text{next}} = 0$ and $Q'_{\text{next}} = 1$
- This is how you **reset**, or **clear**, the latch to 0. The R input stands for "reset"
- Again, it can take two gate delays before a change in R propagates to the output Q'_{next}



$$\begin{aligned}Q_{\text{next}} &= (R + Q'_{\text{current}})' \\Q'_{\text{next}} &= (S + Q_{\text{current}})'\end{aligned}$$

SR latches are memories!

- This little table shows that our latch provides everything we need in a memory: we can set it, reset it, and remember the current value.
- The output Q represents the data stored in the latch. It is sometimes called the **state** of the latch.
- We can expand the table above into a **state table**, which explicitly shows that the *next* values of Q and Q' depend on their *current* values, as well as on the inputs S and R.

S	R	Q
0	0	No change
0	1	0 (reset)
1	0	1 (set)

Inputs		Current		Next	
S	R	Q	Q'	Q	Q'
0	0	0	1	0	1
0	0	1	0	1	0
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	1	1	0
1	0	1	0	1	0

SR latches are sequential!

- Notice that for inputs $SR = 00$, the next value of Q could be either 0 or 1, depending on the current value of Q
- So the same inputs can yield different outputs, depending on whether the latch was previously set or reset
- This is very different from the combinational circuits that we've seen so far, where the same inputs always yield the same outputs.

S	R	Q
0	0	No change
0	1	0 (reset)
1	0	1 (set)

Inputs		Current		Next	
S	R	Q	Q'	Q	Q'
0	0	0	1	0	1
0	0	1	0	1	0
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	1	1	0
1	0	1	0	1	0

What about SR = 11?

- Both Q_{next} and Q'_{next} will become 0
- This contradicts the assumption that Q and Q' are always complements
- Another problem is what happens if we then make S = 0 and R = 0 together.

$$Q_{\text{next}} = (0 + 0)' = 1$$

$$Q'_{\text{next}} = (0 + 0)' = 1$$

- But these new values go back into the NOR gates, and in the next step we get:

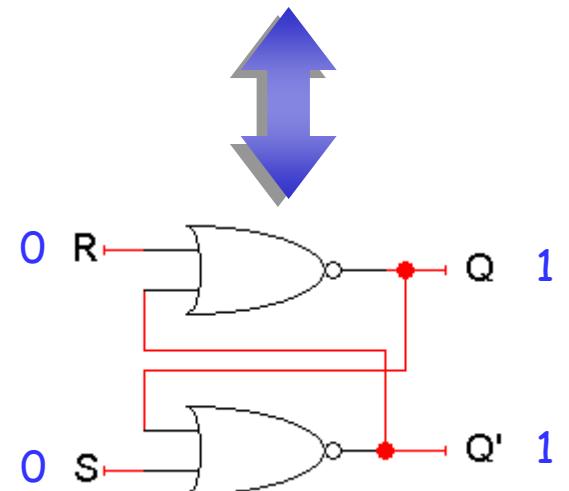
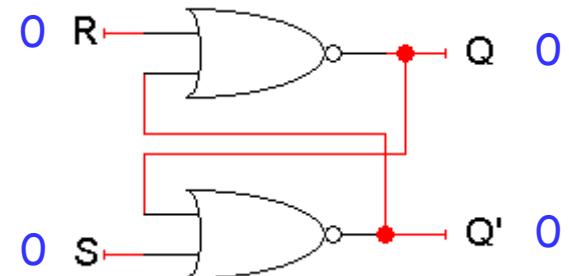
$$Q_{\text{next}} = (0 + 1)' = 0$$

$$Q'_{\text{next}} = (0 + 1)' = 0$$

- The circuit enters an infinite loop, where Q and Q' cycle between 0 and 1 forever
- This is actually the worst case, but the moral is don't ever set SR=11!

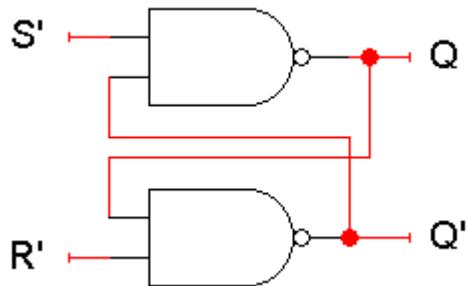
$$Q_{\text{next}} = (R + Q'_{\text{current}})'$$

$$Q'_{\text{next}} = (S + Q_{\text{current}})'$$



$S'R'$ latch

- There are several varieties of latches
- You can use NAND instead of NOR gates to get a $S'R'$ latch

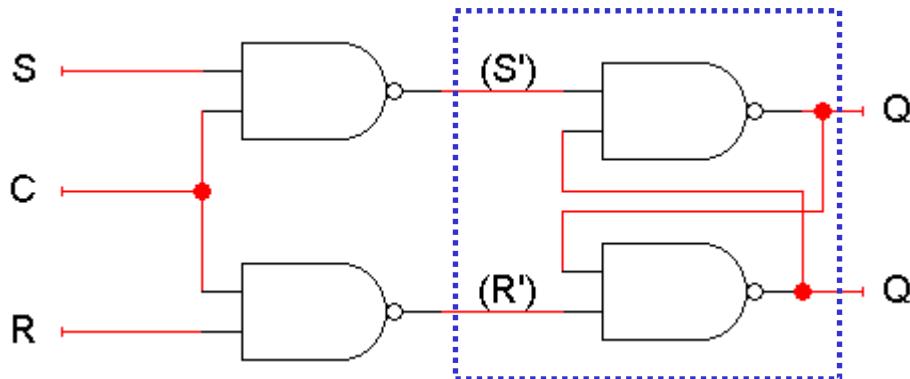


S'	R'	Q
1	1	No change
1	0	0 (reset)
0	1	1 (set)
0	0	Avoid!

- This is just like an SR latch, but with inverted inputs, as you can see from the table
- You can derive this table by writing equations for the outputs in terms of the inputs and the current state, just as we did for the SR latch

An SR latch with a control input

- Here is an SR latch with a control input C

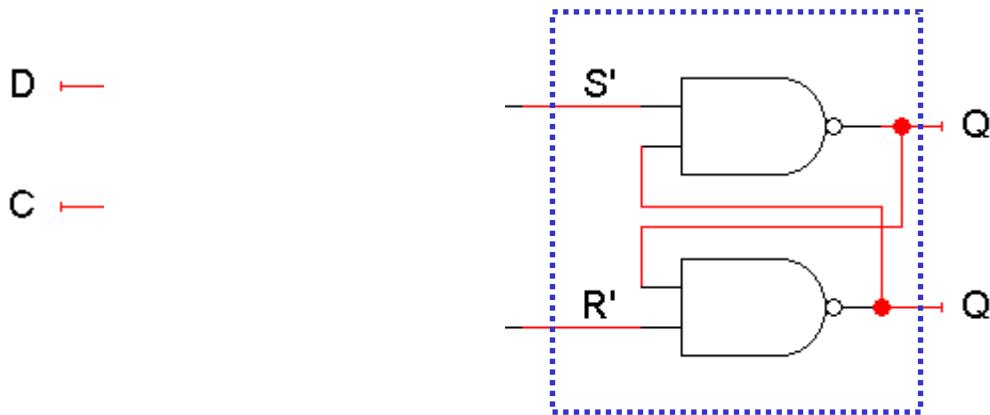


C	S	R	S'	R'	Q
0	x	x	1	1	No change
1	0	0	1	1	No change
1	0	1	1	0	0 (reset)
1	1	0	0	1	1 (set)
1	1	1	0	0	Avoid!

- Notice the hierarchical design!
 - The dotted blue box is the $S'R'$ latch from the previous slide
 - The additional NAND gates are simply used to generate the correct inputs for the $S'R'$ latch
- The control input acts just like an enable

D latch

- Finally, a D latch is based on an S'R' latch. The additional gates generate the S' and R' signals, based on inputs D ("data") and C ("control")
 - When $C = 0$, S' and R' are both 1, so the state Q does not change
 - When $C = 1$, the latch output Q will equal the input D
- No more messing with one input for set and another input for reset!

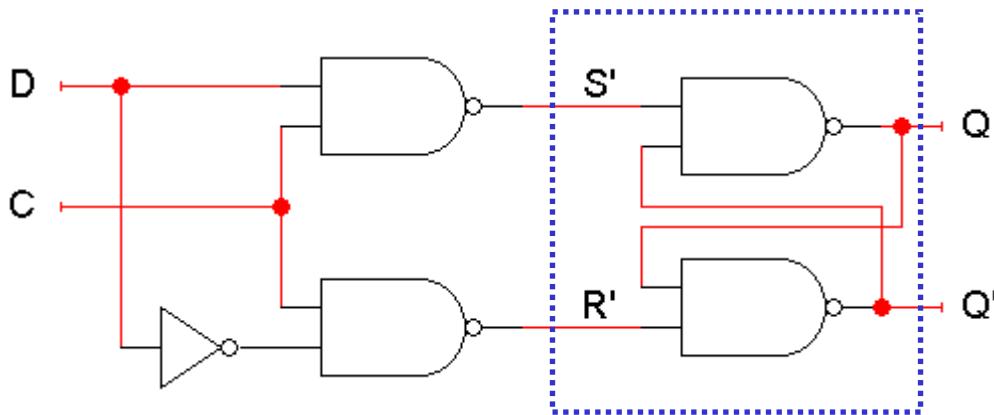


C	D	Q
0	x	No change
1	0	0
1	1	1

- Also, this latch has no "bad" input combinations to avoid. Any of the four possible assignments to C and D are valid.

D latch

- Finally, a D latch is based on an S'R' latch. The additional gates generate the S' and R' signals, based on inputs D ("data") and C ("control")
 - When $C = 0$, S' and R' are both 1, so the state Q does not change
 - When $C = 1$, the latch output Q will equal the input D
- No more messing with one input for set and another input for reset!

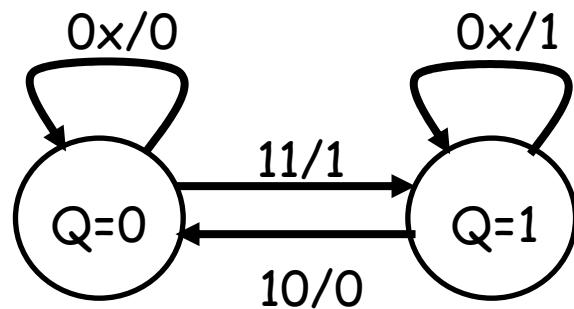


C	D	Q
0	x	No change
1	0	0
1	1	1

- Also, this latch has no "bad" input combinations to avoid. Any of the four possible assignments to C and D are valid.

Sequential circuits and state diagrams

- To describe combinational circuits, we used Boolean expressions and truth tables. With sequential circuits, we can still use expression and tables, but we can also use another form called a **state diagram**
- We draw one node for each state that the circuit can be in. Latches have only two states: $Q=0$ and $Q=1$
- Arrows between nodes are labeled with "input/output" and indicate how the circuit changes states and what its outputs are. In this case the state and the output are the same
- Here's a state diagram for a D latch with inputs D and C



Using latches in real life

- We can connect some latches, acting as memory, to an ALU



- Let's say these latches contain some value that we want to increment
 - The ALU should read the current latch value
 - It applies the " $G = X + 1$ " operation
 - The incremented value is stored back into the latches
- At this point, we have to stop the cycle, so the latch value doesn't get incremented again by accident
- One convenient way to break the loop is to disable the latches

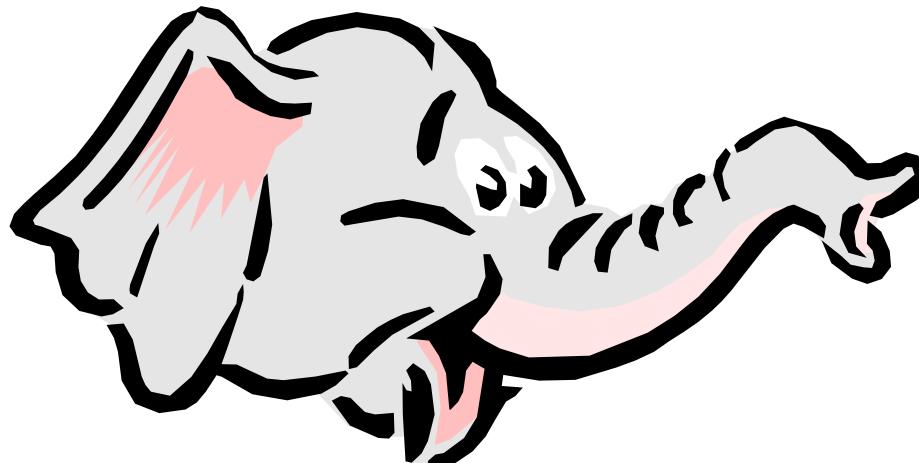
The problem with latches



- The problem is exactly *when* to disable the latches. You have to wait long enough for the ALU to produce its output, but no longer
 - But different ALU operations have different delays. For instance, arithmetic operations might go through an adder, whereas logical operations don't
 - Changing the ALU implementation, such as using a carry-lookahead adder instead of a ripple-carry adder, also affects the delay
- In general, it's very difficult to know how long operations take, and how long latches should be enabled for

Summary

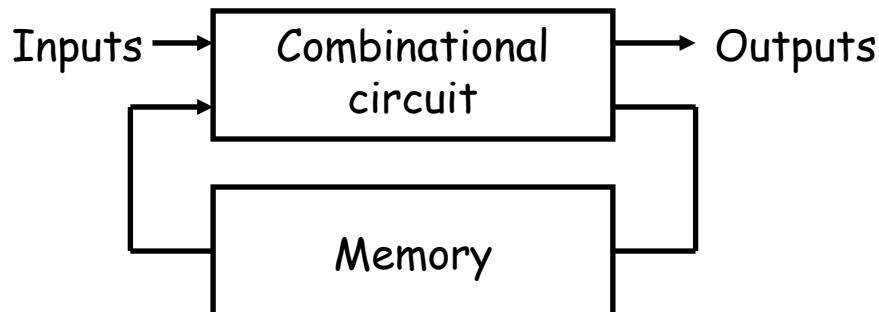
- A sequential circuit has memory. It may respond differently to the same inputs, depending on its current state
- Memories can be created by making circuits with feedback
 - Latches are the simplest memory units, storing individual bits
 - It's difficult to control the timing of latches in a larger circuit
- Next, we'll improve upon latches with **flip-flops**, which change state only at well-defined times. We will then use flip-flops to build all of our sequential circuits



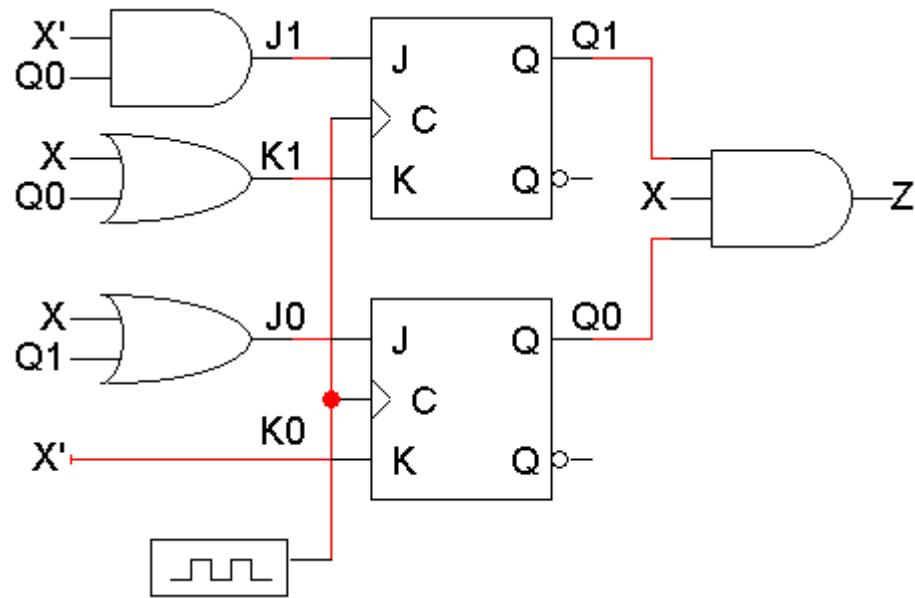
Sequential Circuit Analysis

Mealy or Moore???

- The output of a sequential circuit can be expressed in two different ways:
 - **Moore model**: Outputs= $f(\text{present state})$
 - **Mealy model**: Outputs: $f(\text{present state, inputs})$



An example sequential circuit

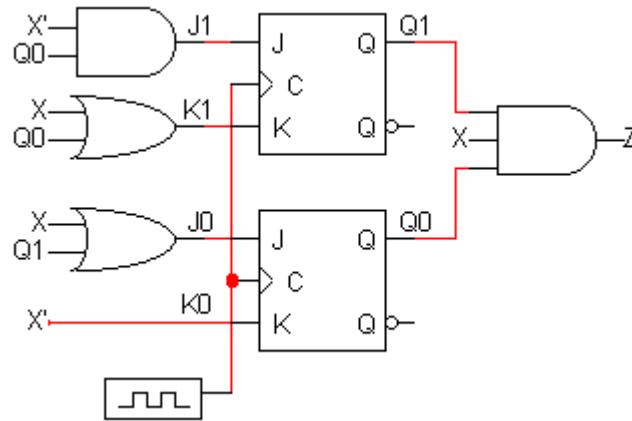


- A sequential circuit with two JK flip-flops
- **State or memory:** Q1Q0
- One input: X; One output: Z

How do you describe a sequential circuit?

- A combinational circuit - Truth table, which shows how the outputs are related to the inputs
- A sequential circuit - **State table**, which shows inputs *and* current states on the left, and outputs *and* next states on the right
 - Need to find the next state of the FFs based on the present state and inputs
 - Need to find the output of the circuit as a function of
 - > current state for a circuit of the Moore model
 - > current state and inputs for a circuit of the Mealy model

State table of example circuit



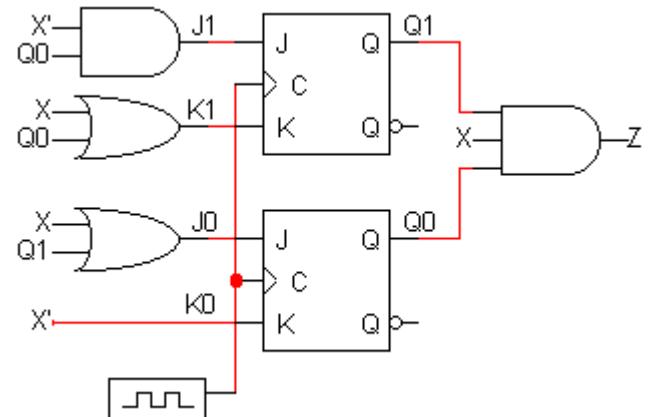
Present State		Inputs X	Next State		Outputs Z
Q_1	Q_0		Q_1	Q_0	
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

The outputs are easy

- From the diagram, you can see that

$$Z = Q_1 Q_0 X$$

Mealy model circuit !!!



Present State		Inputs X	Next State		Outputs Z
Q_1	Q_0		Q_1	Q_0	
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

Flip-flop input equations

- Finding the next states is harder

Step 1:

Find Boolean expressions for the flip-flop inputs
i.e., How do the inputs (say, J & K) to the flip-flops depend on the current state and input

Step 2:

Use these expressions to find the actual flip-flop input values for each possible combination of present states and inputs
i.e., Fill in the state table (with new intermediate columns)

Step 3:

Use flip-flop characteristic tables or equations to find the next states, based on the flip-flop input values and the present states

Step 1: Flip-flop input equations

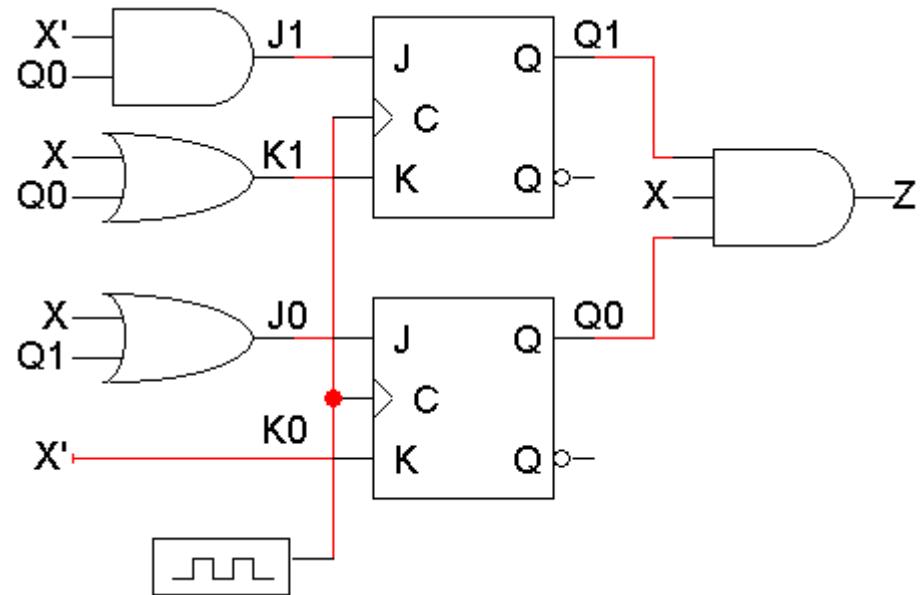
- For our example, the **flip-flop input equations** are:

$$J_1 = X' Q_0$$

$$K_1 = X + Q_0$$

$$J_0 = X + Q_1$$

$$K_0 = X'$$



Step 2: Flip-flop input values

- With these equations, we can make a table showing J_1 , K_1 , J_0 and K_0 for the different combinations of present state Q_1Q_0 and input X

$$J_1 = X' Q_0$$

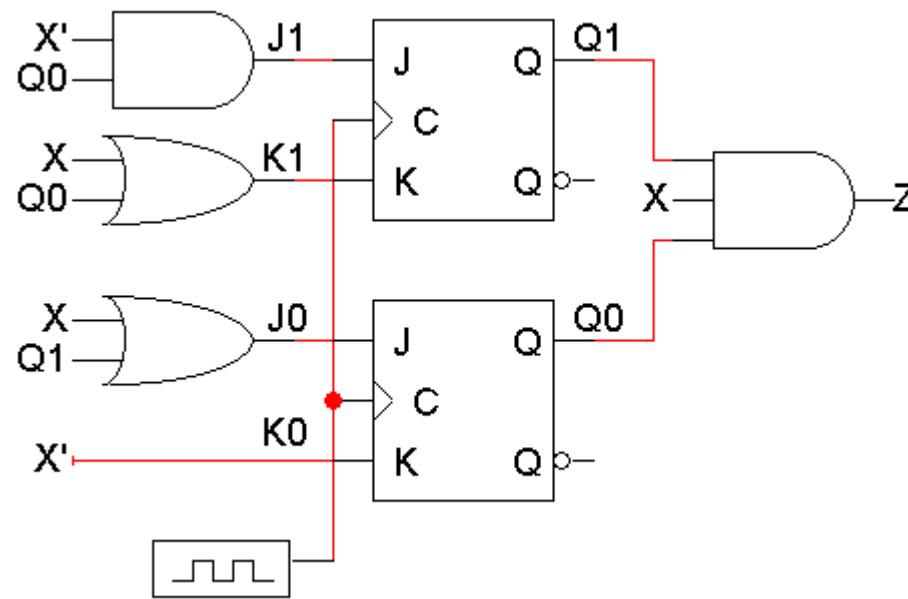
$$K_1 = X + Q_0$$

$$J_0 = X + Q_1$$

$$K_0 = X'$$

Present State		Inputs X	Flip-flop Inputs			
Q_1	Q_0		J_1	K_1	J_0	K_0
0	0	0	0	0	0	1
0	0	1	0	1	1	0
0	1	0	1	1	0	1
0	1	1	0	1	1	0
1	0	0	0	0	1	1
1	0	1	0	1	1	0
1	1	0	1	1	1	1
1	1	1	0	1	1	0

Step 2: Flip-flop input values



Present State		Inputs X	Flip-flop Inputs			
Q_1	Q_0		J_1	K_1	J_0	K_0
0	0	0	0	0	0	1
0	0	1	0	1	1	0
0	1	0	1	1	0	1
0	1	1	0	1	1	0
1	0	0	0	0	1	1
1	0	1	0	1	1	0
1	1	0	1	1	1	1
1	1	1	0	1	1	0

Step 3: Find the next states

- Finally, use the JK flip-flop characteristic tables or equations to find the next state of *each* flip-flop, based on its present state and inputs
- The general JK flip-flop characteristic equation is:

$$Q(t+1) = K'Q(t) + JQ'(t)$$

- In our example circuit, we have two JK flip-flops, so we have to apply this equation to *each* of them:

$$Q_1(t+1) = K_1'Q_1(t) + J_1Q_1'(t)$$

$$Q_0(t+1) = K_0'Q_0(t) + J_0Q_0'(t)$$

- We can also determine the next state for each input/current state combination directly from the characteristic table

J	K	Q(t+1)	Operation
0	0	Q(t)	No change
0	1	0	Reset
1	0	1	Set
1	1	Q'(t)	Complement

Step 3 concluded

- The next states for Q_1 and Q_0 , are calculated using these equations:

$$\longrightarrow Q_1(t+1) = K_1'Q_1(t) + J_1Q_1'(t)$$

$$Q_0(t+1) = K_0'Q_0(t) + J_0Q_0'(t)$$

Present State		Inputs X	FF Inputs				Next State	
Q_1	Q_0		J_1	K_1	J_0	K_0	Q_1	Q_0
0	0	0	0	0	0	1		
0	0	1	0	1	1	0		
0	1	0	1	1	0	1	1	
0	1	1	0	1	1	0		
1	0	0	0	0	1	1		
1	0	1	0	1	1	0		
1	1	0	1	1	1	1		
1	1	1	0	1	1	0		

Step 3 concluded

- Using the characteristic equations:

$$Q_1(t+1) = K_1'Q_1(t) + J_1Q_1'(t)$$

$$Q_0(t+1) = K_0'Q_0(t) + J_0Q_0'(t)$$

Or the characteristic table

J	K	$Q(t+1)$
0	0	$Q(t)$
0	1	0
1	0	1
1	1	$Q'(t)$

Present State		Inputs X	FF Inputs				Next State		
Q_1	Q_0		J_1	K_1	J_0	K_0	Q_1	Q_0	
0	0	0	0	0	0	1	0		
0	0		0	1	1	0			
0	1		1	1	0	1			
0	1		0	1	1	0			
1	0	0	0	0	1	1			
1	0	1	0	1	1	0			
1	1	0	1	1	1	1	0		
1	1	1	0	1	1	0			

Step 3 concluded

- Finally, here are the next states for Q_1 and Q_0 , using these equations:

$$Q_1(t+1) = K_1'Q_1(t) + J_1Q_1'(t)$$

$$Q_0(t+1) = K_0'Q_0(t) + J_0Q_0'(t)$$

Present State		Inputs X	FF Inputs				Next State	
Q_1	Q_0		J_1	K_1	J_0	K_0	Q_1	Q_0
0	0	0	0	0	0	1	0	0
0	0	1	0	1	1	0	0	1
0	1	0	1	1	0	1	1	0
0	1	1	0	1	1	0	0	1
1	0	0	0	0	1	1	1	1
1	0	1	0	1	1	0	0	1
1	1	0	1	1	1	1	0	0
1	1	1	0	1	1	0	0	1

Getting the state table columns straight

- The table starts with Present State and Inputs
 - Present State and Inputs yield FF Inputs
 - Present State and FF Inputs yield Next State, based on the flip-flop characteristic tables
 - Present State and Inputs yield Output

Present State		Inputs X	FF Inputs				Next State		Outputs Z
Q ₁	Q ₀		J ₁	K ₁	J ₀	K ₀	Q ₁	Q ₀	
0	0	0	0	0	0	1	0	0	0
0	0	1	0	1	1	0	0	1	0
0	1	0	1	1	0	1	1	0	0
0	1	1	0	1	1	0	0	1	0
1	0	0	0	0	1	1	1	1	0
1	0	1	0	1	1	0	0	1	0
1	1	0	1	1	1	1	0	0	0
1	1	1	0	1	1	0	0	1	1

A different look

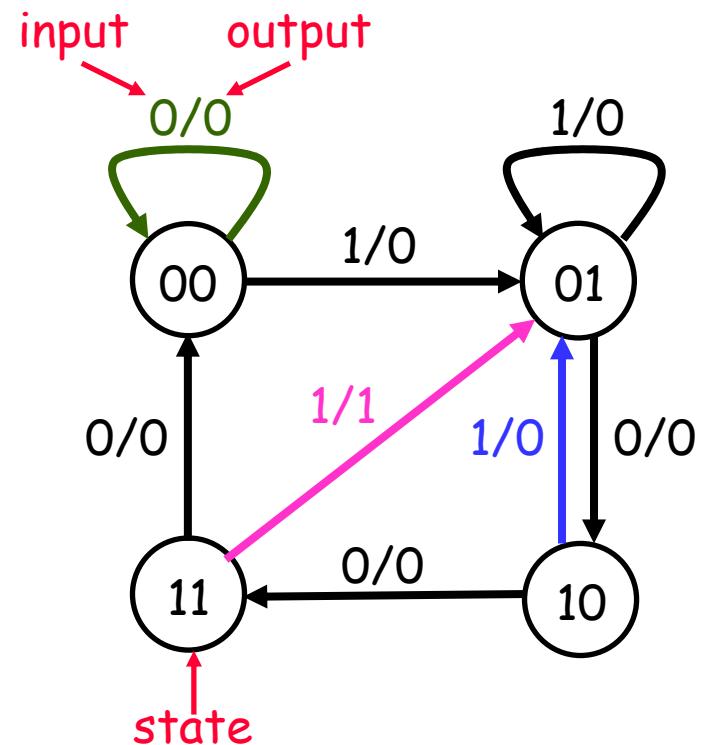
Present State		Inputs X	Next State		Outputs Z
Q ₁	Q ₀		Q ₁	Q ₀	
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	1	0	0	0	0
1	1	1	0	1	1

Present State		Next State				Output Z	
		Input X= 0		Input X= 1		X= 0	X= 1
Q1	Q0	0	0	0	1	0	0
0	0	1	0	0	1	0	0
1	0	1	1	0	1	0	0
1	1	0	0	0	1	0	1

State diagrams (Mealy model)

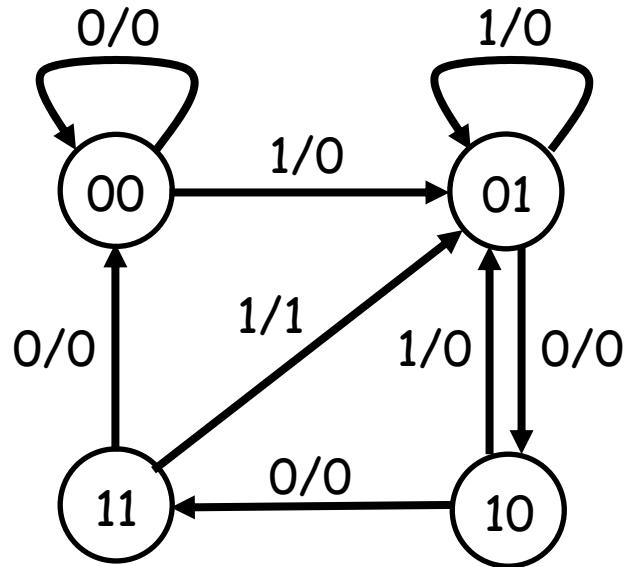
- We can also represent the state table graphically with a state diagram
- A diagram corresponding to our example state table is shown below

Present State		Inputs	Next State		Outputs
Q_1	Q_0	X	Q_1	Q_0	Z
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	0	1	0
1	0	0	1	1	0
1	0	1	0	1	0
1	1	0	0	0	0
1	1	1	0	1	1

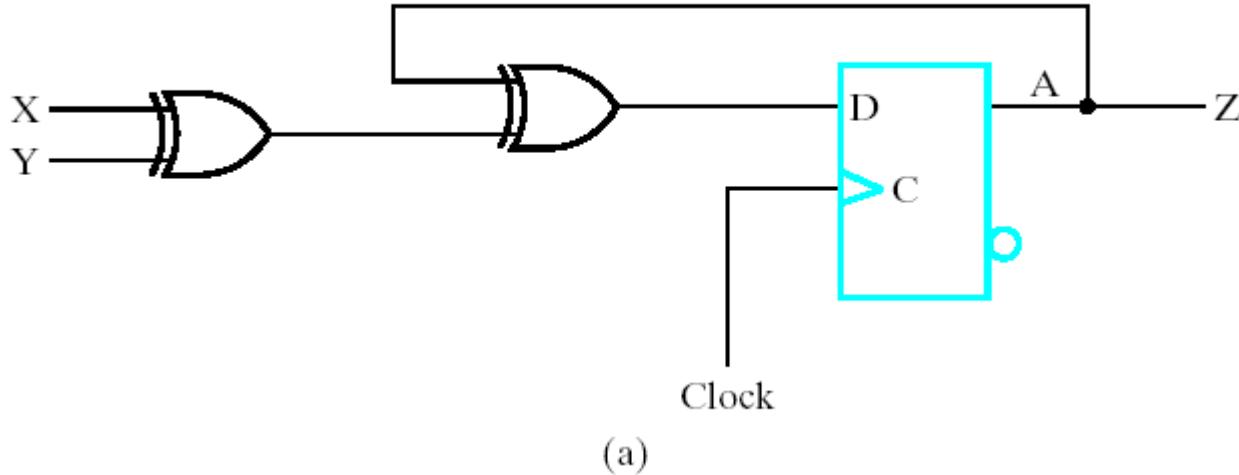


Sizes of state diagrams

- Always check the size of your state diagrams
 - If there are n flip-flops, there should be 2^n nodes in the diagram
 - If there are m inputs, then each node will have 2^m outgoing arrows
- In our example,
 - We have two flip-flops, and thus four states or nodes.
 - There is one input, so each node has two outgoing arrows.

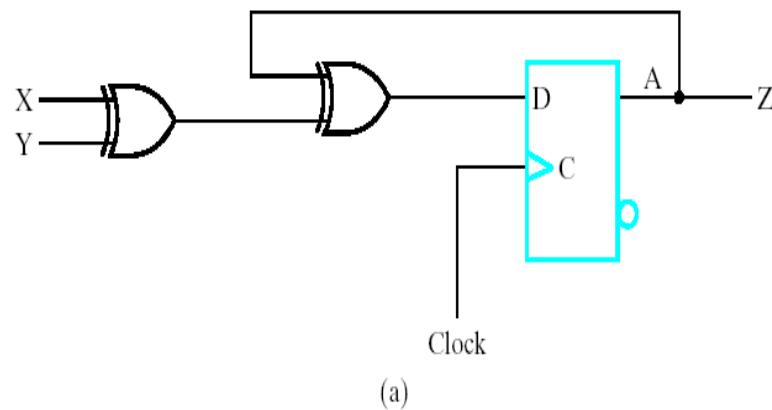


Moore type circuit



- Two inputs: X and Y ; One output: Z
- One state: A
- Note that $Z = A$, just a function of the current state

State table (Moore)



Present state	Inputs		Next state	Output
	A	X Y		
0	0	0 0	0	0
0	0	0 1	1	0
0	1	0 0	1	0
0	1	0 1	0	0
1	0	0 0	1	1
1	0	0 1	0	1
1	1	0 0	0	1
1	1	0 1	1	1

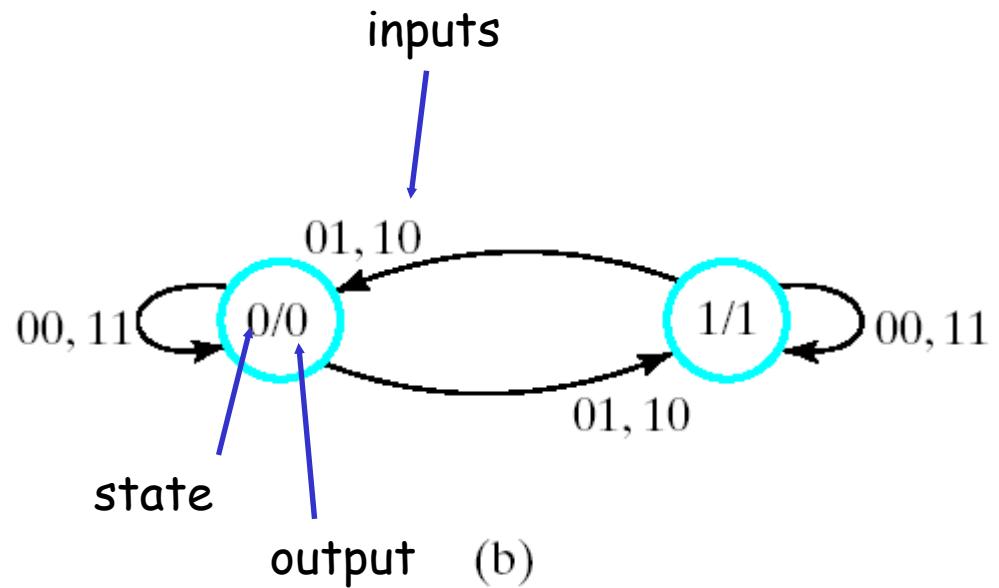
(b) State table

Present State A	Next State Inputs XY				Output Z
	00	01	10	11	
	0	1	1	0	0
1	1	0	0	1	1

State diagram (Moore)

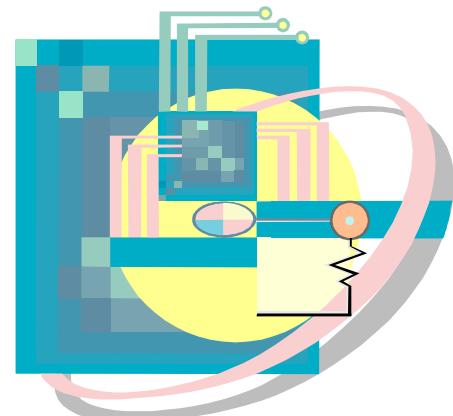
Present state	Inputs		Next state	Output
A	X	Y	A	Z
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(b) State table

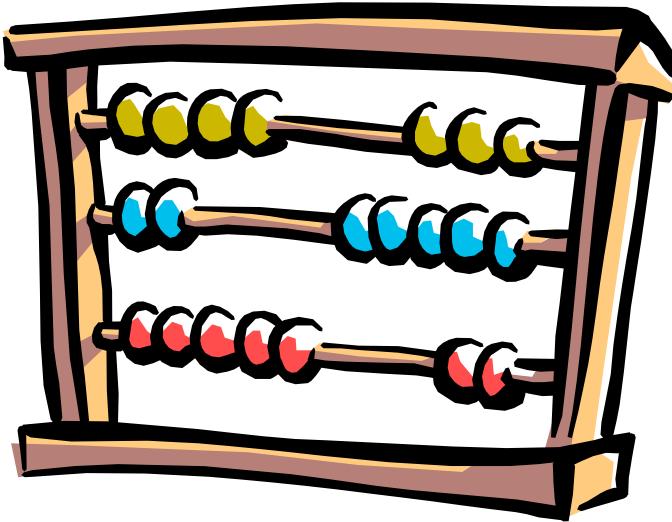


Sequential circuit analysis summary

- To analyze sequential circuits, you have to:
 - Find Boolean expressions for the outputs of the circuit and the flip-flop inputs
 - Use these expressions to fill in the output and flip-flop input columns in the state table
 - Finally, use the characteristic equation or characteristic table of the flip-flop to fill in the next state columns.
- The result of sequential circuit analysis is a state table or a state diagram describing the circuit



Counters



- We'll look at different kinds of **counters** and discuss how to build them
- These are not only examples of sequential analysis and design, but also real devices used in larger circuits

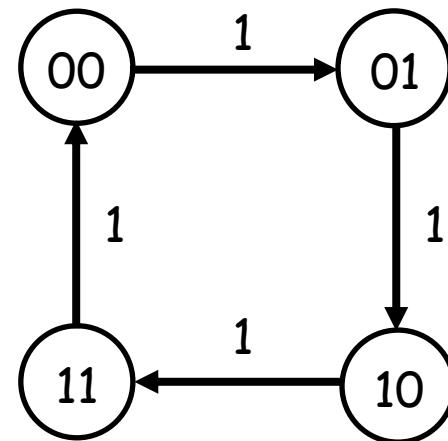
Introducing counters

- Counters are a specific type of sequential circuit
- The state serves as the “output” (Moore)
- A counter that follows the binary number sequence is called a **binary counter**
 - n-bit binary counter: n flip-flops, count in binary from 0 to $2^n - 1$
- Counters are available in two types:
 - Synchronous Counters
 - Ripple Counters
- **Synchronous Counters:**
 - A common clock signal is connected to the C input of each flip-flop

Synchronous Binary Up Counter

- The output value increases by one on each clock cycle
- After the largest value, the output “wraps around” back to 0
- Using two bits, we’d get something like this:

Present State		Next State	
A	B	A	B
0	0	0	1
0	1	1	0
1	0	1	1
1	1	0	0



What good are counters?

- Counters can act as simple clocks to keep track of "time"
- You may need to record how many times something has happened
 - How many bits have been sent or received?
 - How many steps have been performed in some computation?
- All processors contain a **program counter**, or **PC**
 - Programs consist of a list of instructions that are to be executed one after another (for the most part)
 - The PC keeps track of the instruction currently being executed
 - The PC increments once on each clock cycle, and the next program instruction is then executed.

Synch Binary Up/Down Counter

- 2-bit Up/Down counter
 - Counter outputs will be 00, 01, 10 and 11
 - There is a single input, X.
 - > $X=0$, the counter counts up
 - > $X=1$, the counter counts down
- We'll need two flip-flops again. Here are the four possible states:

00

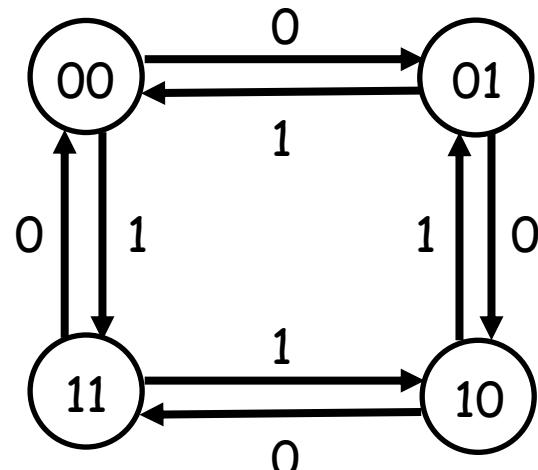
01

11

10

The complete state diagram and table

- Here's the complete state diagram and state table for this circuit



Present State		Inputs X	Next State	
Q_1	Q_0		Q_1	Q_0
0	0	0	0	1
0	0	1	1	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	1
1	1	0	0	0
1	1	1	1	0

D flip-flop inputs

- If we use D flip-flops, then the D inputs will just be the same as the desired next states
- Equations for the D flip-flop inputs are shown at the right
- Why does $D_0 = Q_0'$ make sense?

Present State		Inputs	Next State	
Q_1	Q_0	X	Q_1	Q_0
0	0	0	0	1
0	0	1	1	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	1
1	0	1	0	1
1	1	0	0	0
1	1	1	1	0

		Q_0		
		0	1	0
Q_1	1	0	1	0
		X		

$$D_1 = Q_1 \oplus Q_0 \oplus X$$

		Q_0		
		1	1	0
Q_1	1	1	0	0
		X		

$$D_0 = Q_0'$$

JK flip-flop inputs

- If we use JK flip-flops instead, then we have to compute the JK inputs for each flip-flop
- Look at the present and desired next state, and use the excitation table on the right

Q(t)	Q(t+1)	J	K
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

Present State		Inputs X	Next State		Flip flop inputs			
Q ₁	Q ₀		Q ₁	Q ₀	J ₁	K ₁	J ₀	K ₀
0	0	0	0	1	0	x	1	x
0	0	1	1	1	1	x	1	x
0	1	0	1	0	1	x	x	1
0	1	1	0	0	0	x	x	1
1	0	0	1	1	x	0	1	x
1	0	1	0	1	x	1	1	x
1	1	0	0	0	x	1	x	1
1	1	1	1	0	x	0	x	1

JK flip-flop input equations

Present State		Inputs X	Next State		Flip flop inputs			
Q ₁	Q ₀		Q ₁	Q ₀	J ₁	K ₁	J ₀	K ₀
0	0	0	0	1	0	x	1	x
0	0	1	1	1	1	x	1	x
0	1	0	1	0	1	x	x	1
0	1	1	0	0	0	x	x	1
1	0	0	1	1	x	0	1	x
1	0	1	0	1	x	1	1	x
1	1	0	0	0	x	1	x	1
1	1	1	1	0	x	0	x	1

- We can then find equations for all four flip-flop inputs, in terms of the present state and inputs. Here, it turns out $J_1 = K_1$ and $J_0 = K_0$

$$J_1 = K_1 = Q_0' X + Q_0 X'$$

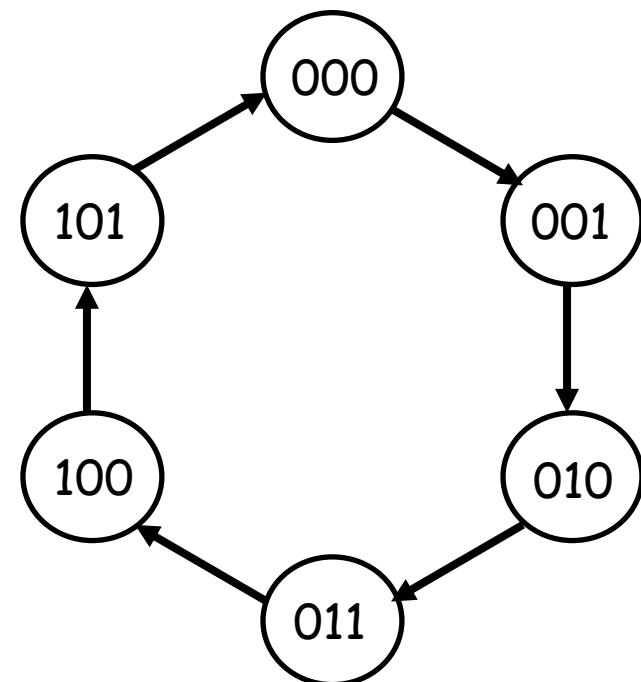
$$J_0 = K_0 = 1$$

- Why does $J_0 = K_0 = 1$ make sense?

Unused states

- The examples shown so far have all had 2^n states, and used n flip-flops. But sometimes you may have unused, leftover states
- For example, here is a state table and diagram for a counter that repeatedly counts from 0 (000) to 5 (101)
- What should we put in the table for the two unused states?

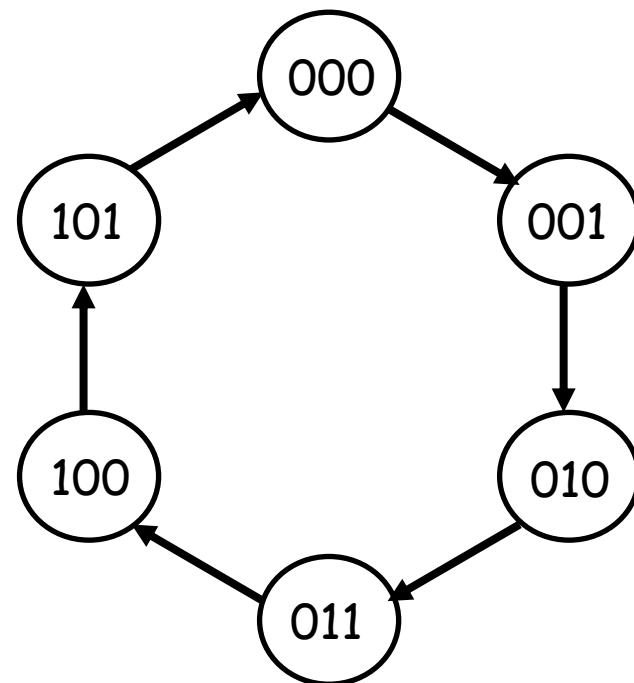
Present State			Next State		
Q_2	Q_1	Q_0	Q_2	Q_1	Q_0
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	0	0	0
1	1	0	?	?	?
1	1	1	?	?	?



Unused states can be don't cares...

- To get the *simpliest* possible circuit, you can fill in don't cares for the next states. This will also result in don't cares for the flip-flop inputs, which can simplify the hardware
- If the circuit somehow ends up in one of the unused states (110 or 111), its behavior will depend on exactly what the don't cares were filled in with

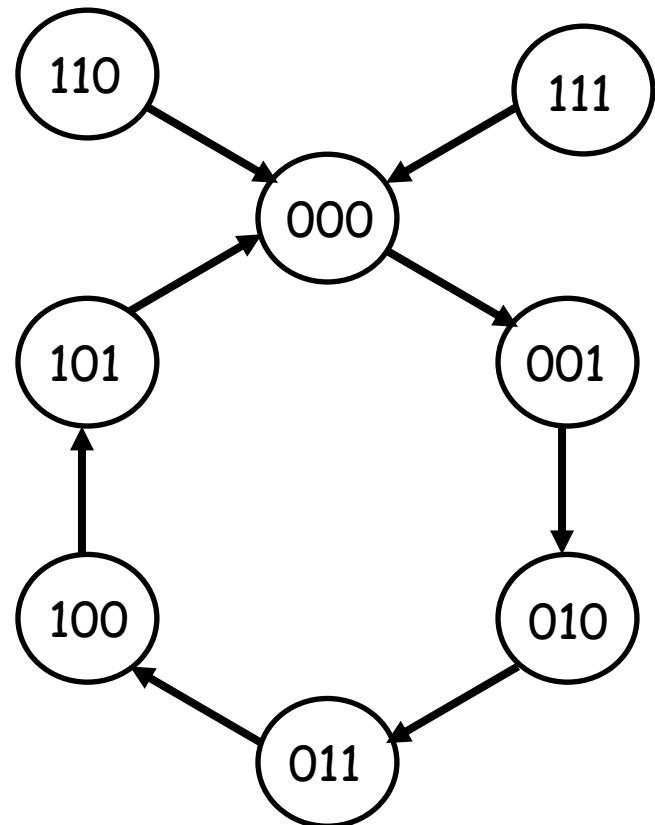
Present State			Next State		
Q_2	Q_1	Q_0	Q_2	Q_1	Q_0
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	0	0	0
1	1	0	x	x	x
1	1	1	x	x	x



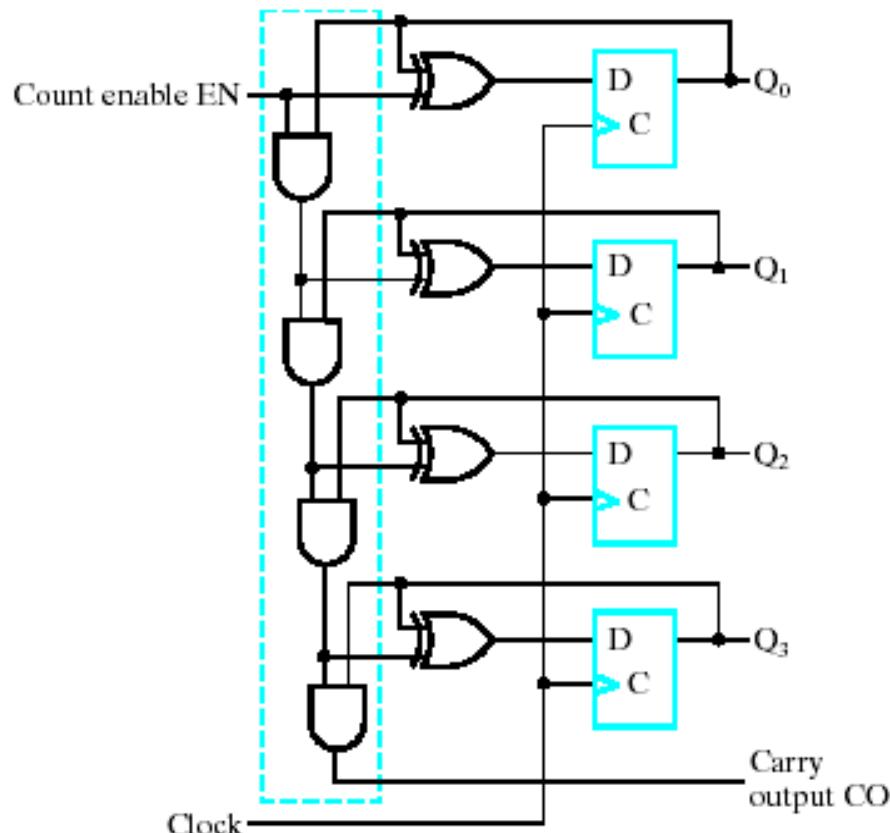
...or maybe you do care

- To get the *safest* possible circuit, you can explicitly fill in next states for the unused states 110 and 111
- This guarantees that even if the circuit somehow enters an unused state, it will eventually end up in a valid state
- This is called a **self-starting counter**

Present State			Next State		
Q_2	Q_1	Q_0	Q_2	Q_1	Q_0
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	0	0	0
1	1	0	0	0	0
1	1	1	0	0	0



4-bit Counter with Serial Gating



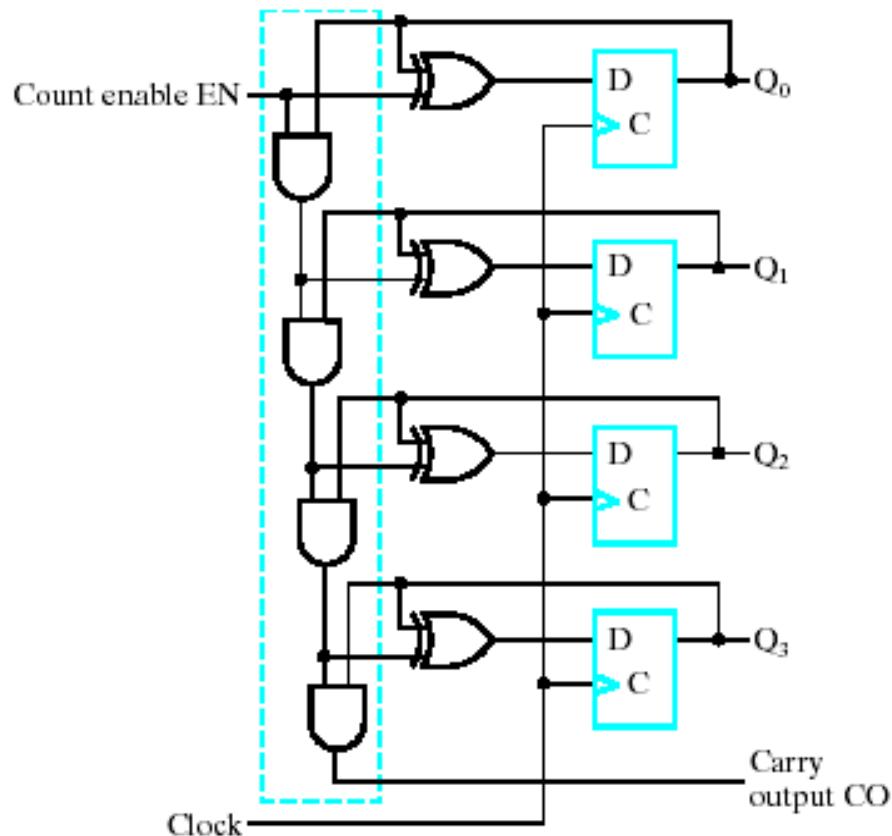
(a) Logic Diagram-Serial Gating

$CO = 1$ when 1111

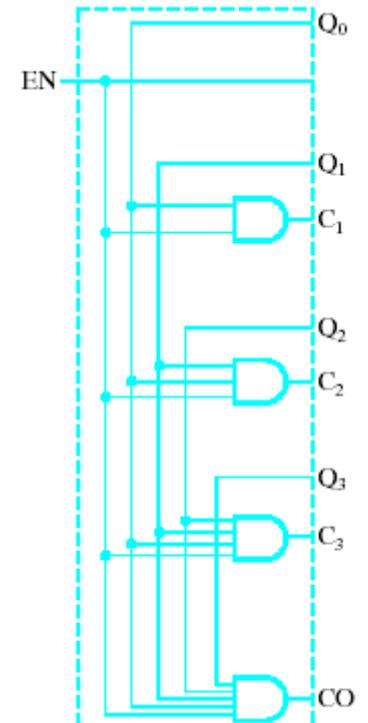
> 4 gate delays

0	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	0	0	0
1	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

4-bit Counter with Parallel Gating

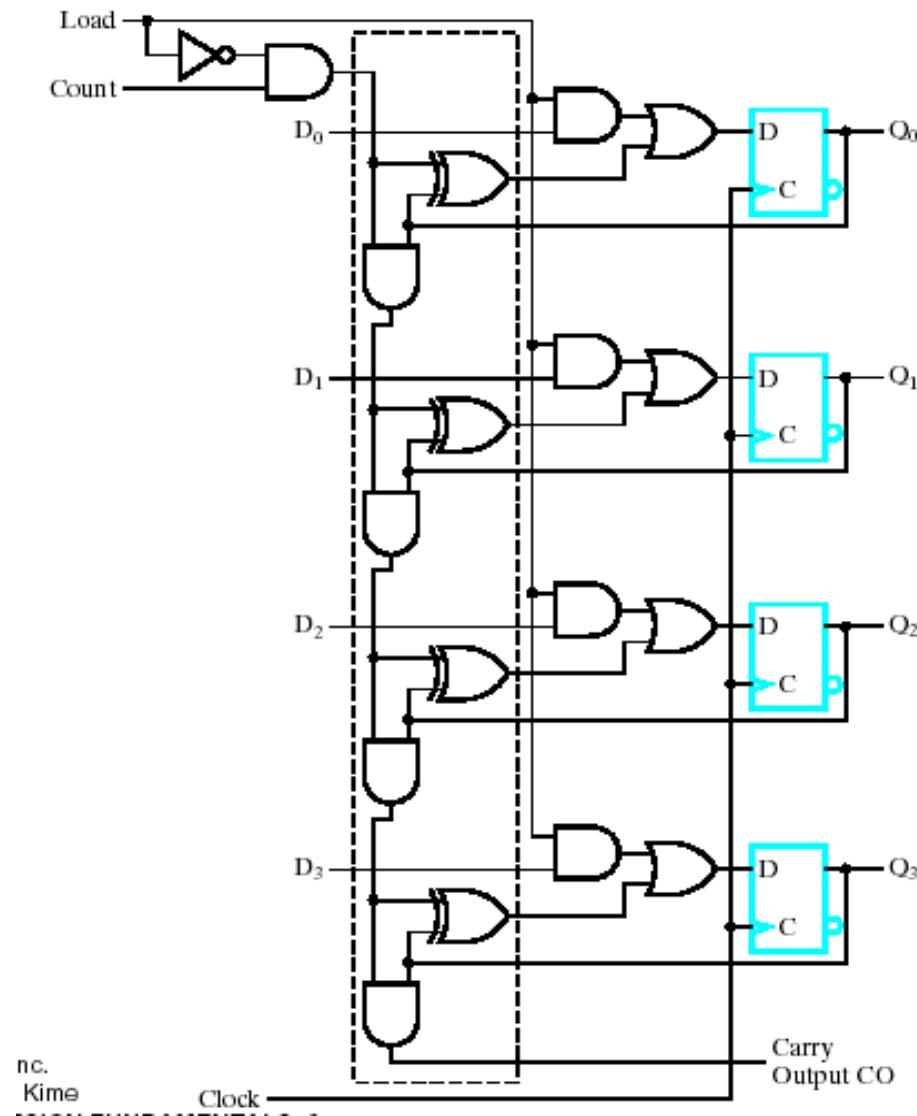


(a) Logic Diagram-Serial Gating



(b) Logic Diagram-Parallel Gating

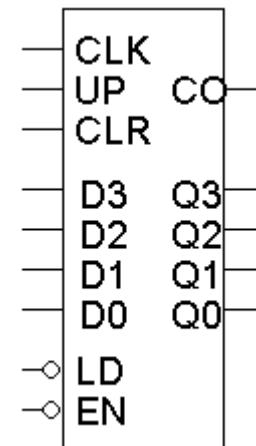
4-bit Binary Counter with Parallel Load



Load	Count	
1	x	Parallel Load
0	0	No Change
0	1	Count

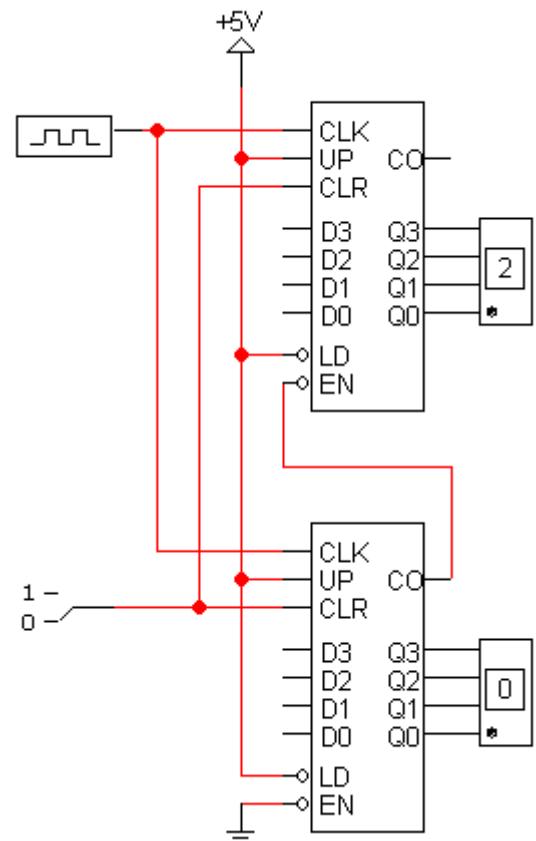
More complex counters

- More complex counters are also possible:
 - It can increment or decrement, by setting the **UP** input to 1 or 0
 - You can immediately (asynchronously) clear the counter to 0000 by setting **CLR = 1**
 - You can specify the counter's next output by setting **D₃-D₀** to any four-bit value and clearing **LD**
 - The active-low **EN** input enables or disables the counter
 - When the counter is disabled, it continues to output the same value without incrementing, decrementing, loading, or clearing
 - The "counter out" **CO** is normally 1, but becomes 0 when the counter reaches its maximum value, 1111



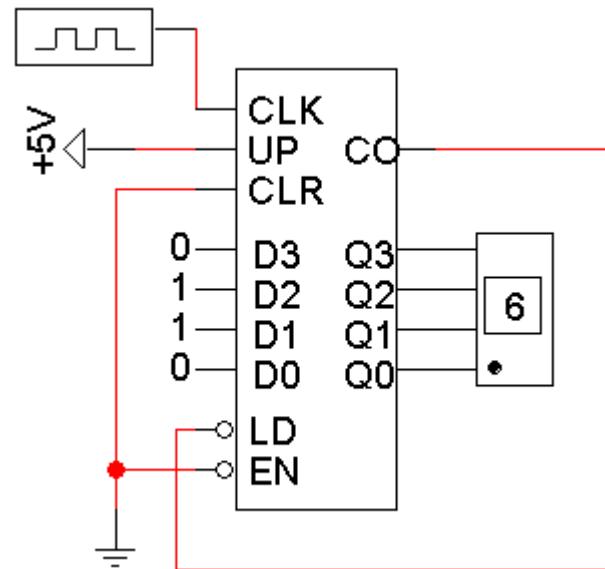
An 8-bit counter

- As you might expect by now, we can use these general counters to build other counters
- Here is an 8-bit counter made from two 4-bit counters
 - The bottom device represents the least significant four bits, while the top counter represents the most significant four bits
 - When the bottom counter reaches 1111 (i.e., when $CO = 0$), it enables the top counter for one cycle
- Other implementation notes:
 - The counters share clock and clear signals
 - Hex displays are used here



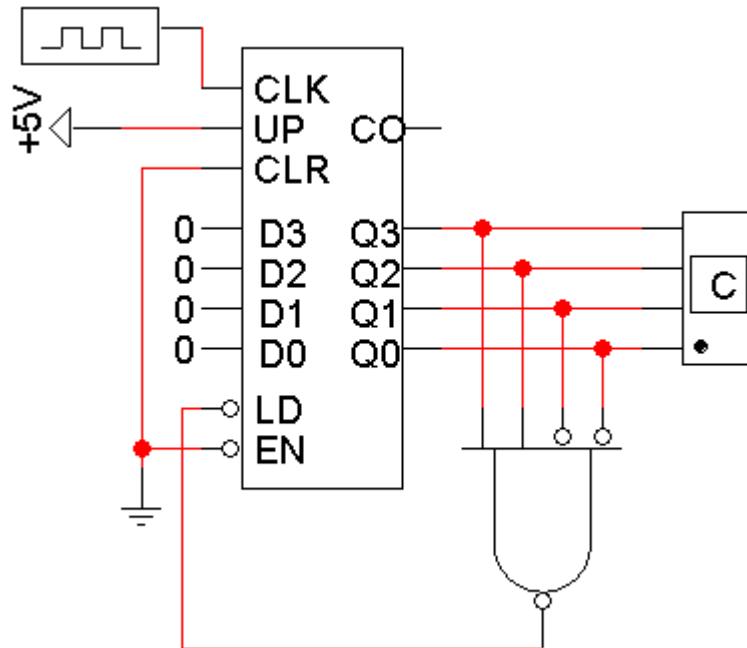
A restricted 4-bit counter

- We can also make a counter that “starts” at some value besides 0000
- In the diagram below, when $CO=0$ the LD signal forces the next state to be loaded from D_3-D_0
- The result is this counter wraps from 1111 to 0110 (instead of 0000)

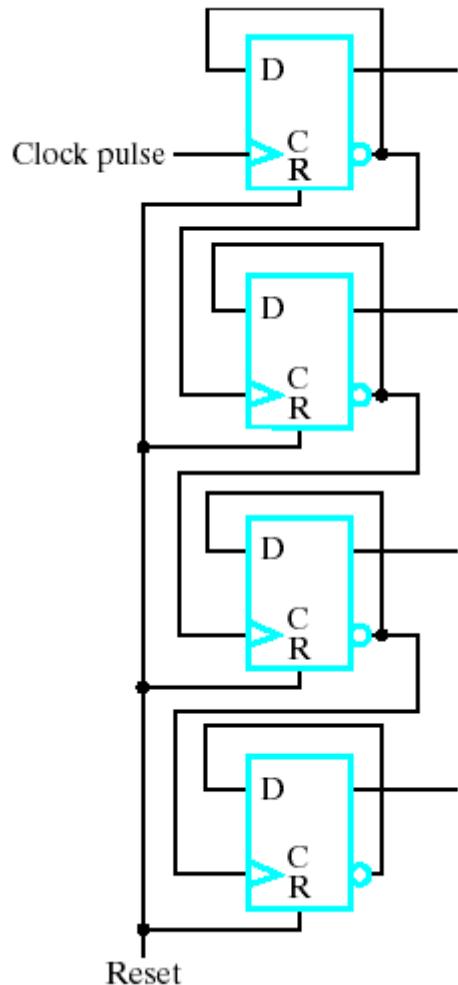


Another restricted counter

- We can also make a circuit that counts up to only 1100, instead of 1111
- Here, when the counter value reaches 1100, the NAND gate forces the counter to load, so the next state becomes 0000



Ripple Counter



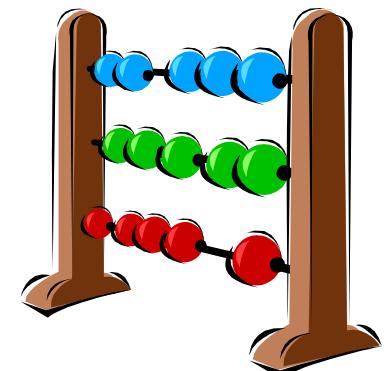
Upward Counting Sequence

Q_3	Q_2	Q_1	Q_0
0	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	0	0	0
1	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

Simple, yet asynchronous circuits !!!

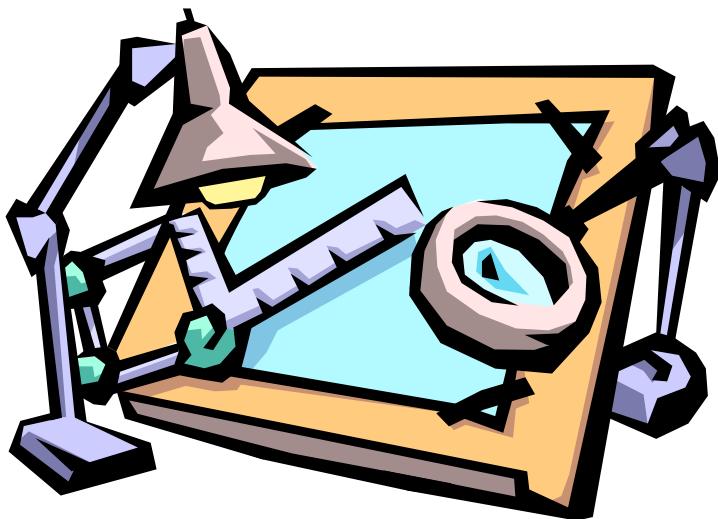
Summary

- Counters serve many purposes in sequential logic design
- There are lots of variations on the basic counter
 - Some can increment or decrement
 - An enable signal can be added
 - The counter's value may be explicitly set
- There are also several ways to make counters
 - You can follow the sequential design principles to build counters from scratch
 - You could also modify or combine existing counter devices



Sequential circuit design

- In **sequential circuit design**, we turn some description into a working circuit
 - We first make a state table or diagram to express the computation
 - Then we can turn that table or diagram into a sequential circuit



Sequence recognizers

- A **sequence recognizer** is a special kind of sequential circuit that looks for a special bit pattern in some input
- The recognizer circuit has only one input, X
 - One bit of input is supplied on every clock cycle
 - This is an easy way to permit arbitrarily long input sequences
- There is one output, Z, which is 1 when the desired pattern is found
- Our example will detect the bit pattern "1001":

Inputs: 1 1 1 0 0 1 1 0 1 0 0 1 0 0 1 1 0 ...
Outputs: 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0 0 ...

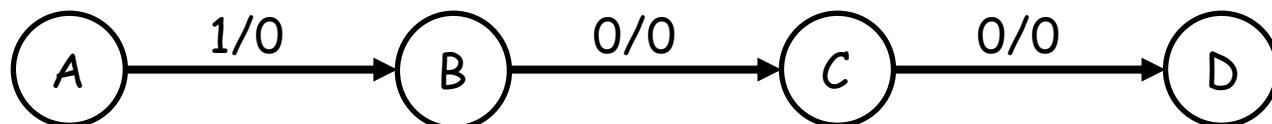
- A sequential circuit is required because the circuit has to "remember" the inputs from previous clock cycles, in order to determine whether or not a match was found

Step 1: Making a state table

- The first thing you have to figure out is precisely how the use of state will help you solve the given problem
 - Make a state table based on the problem statement. The table should show the present states, inputs, next states and outputs
 - Sometimes it is easier to first find a state diagram and then convert that to a table
- This is usually the most difficult step. Once you have the state table, the rest of the design procedure is the same for all sequential circuits

A basic state diagram

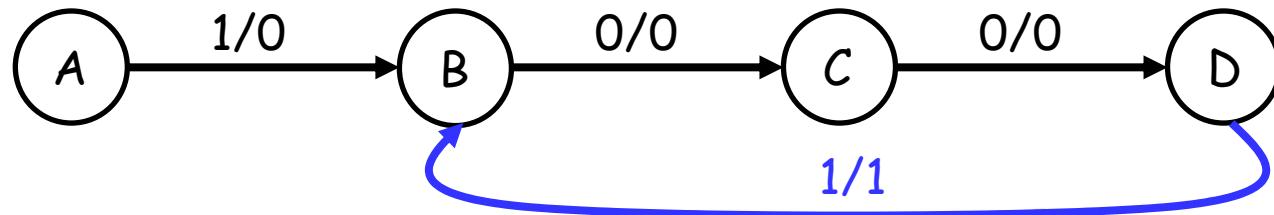
- What state do we need for the sequence recognizer?
 - We have to “remember” inputs from previous clock cycles
 - For example, if the previous three inputs were 100 and the current input is 1, then the output should be 1
 - In general, we will have to remember occurrences of parts of the desired pattern—in this case, 1, 10, and 100
- We'll start with a basic state diagram:



State	Meaning
A	None of the desired pattern (1001) has been input yet.
B	We've already seen the first bit (1) of the desired pattern.
C	We've already seen the first two bits (10) of the desired pattern.
D	We've already seen the first three bits (100) of the desired pattern.

Overlapping occurrences of the pattern

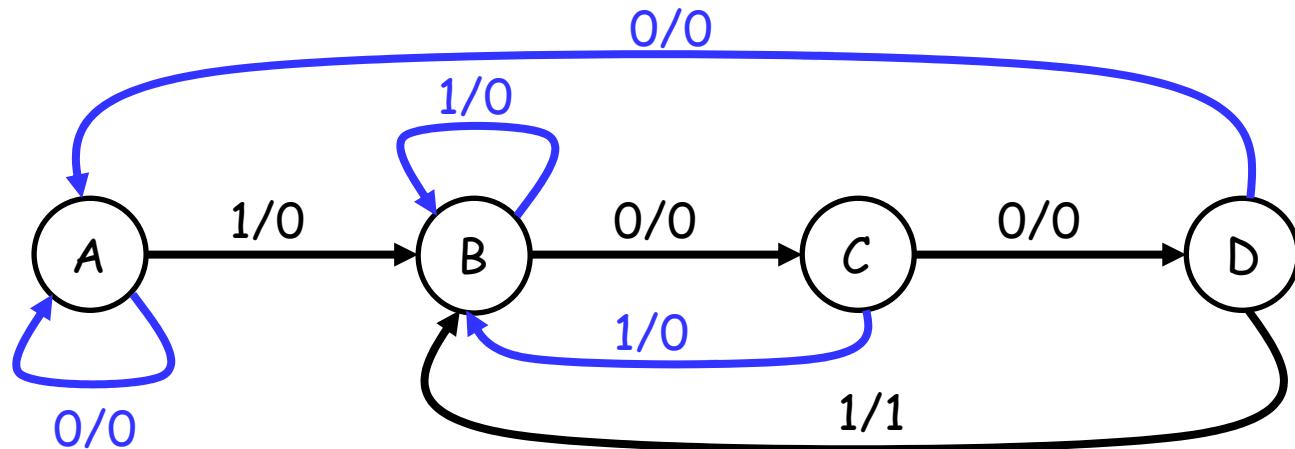
- What happens if we're in state D (the last three inputs were 100), and the current input is 1?
 - The output should be a 1, because we've found the desired pattern
 - But this last 1 could also be the start of another occurrence of the pattern! For example, 1001001 contains *two* occurrences of 1001
 - To detect overlapping occurrences of the pattern, the next state should be B.



State	Meaning
A	None of the desired pattern (1001) has been input yet.
B	We've already seen the first bit (1) of the desired pattern.
C	We've already seen the first two bits (10) of the desired pattern.
D	We've already seen the first three bits (100) of the desired pattern.

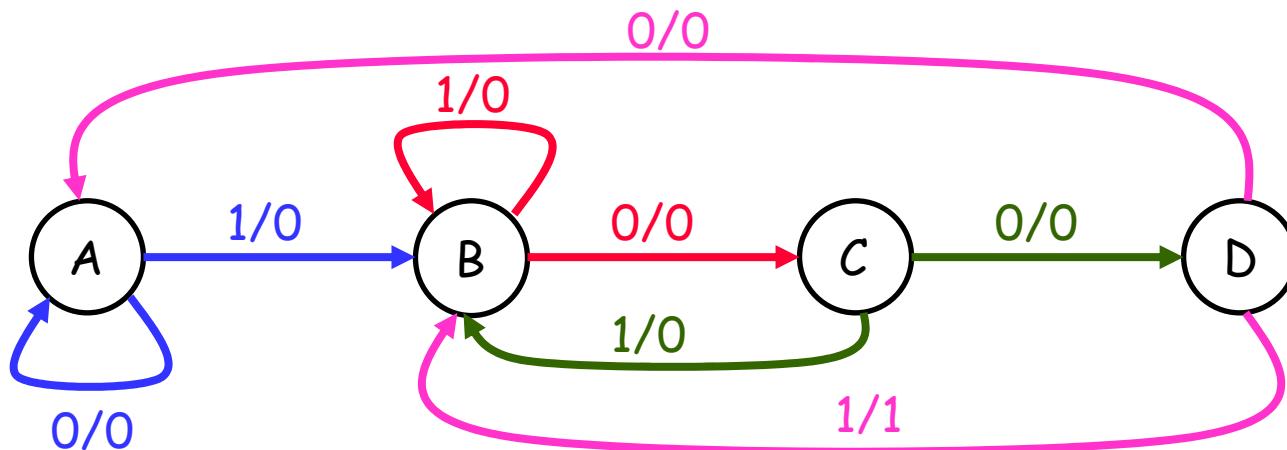
Filling in the other arrows

- Two outgoing arrows for each node, to account for the possibilities of $X=0$ and $X=1$
- The remaining arrows we need are shown in blue. They also allow for the correct detection of overlapping occurrences of 1001.

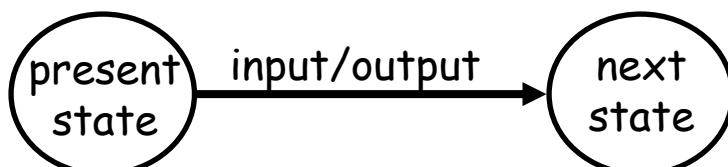


State	Meaning
A	None of the desired pattern (1001) has been input yet.
B	We've already seen the first bit (1) of the desired pattern.
C	We've already seen the first two bits (10) of the desired pattern.
D	We've already seen the first three bits (100) of the desired pattern.

Finally, making the state table



Remember how the state diagram arrows correspond to rows of the state table:



Present State	Input	Next State	Output
A	0	A	0
A	1	B	0
B	0	C	0
B	1	B	0
C	0	D	0
C	1	B	0
D	0	A	0
D	1	B	1

Sequential circuit design procedure

Step 1:

Make a state table based on the problem statement. The table should show the present states, inputs, next states and outputs. (It may be easier to find a state diagram first, and then convert that to a table)

Step 2:

Assign binary codes to the states in the state table, if you haven't already.
If you have n states, your binary codes will have at least
 $\lceil \log_2 n \rceil$ digits, and your circuit will have at least $\lceil \log_2 n \rceil$ flip-flops

Step 3:

For each flip-flop and each row of your state table, find the flip-flop input values that are needed to generate the next state from the present state.
You can use flip-flop excitation tables here.

Step 4:

Find simplified equations for the flip-flop inputs and the outputs.

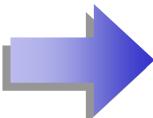
Step 5:

Build the circuit!

Step 2: Assigning binary codes to states

- We have four states ABCD, so we need at least two flip-flops Q_1Q_0
- The easiest thing to do is represent state A with $Q_1Q_0 = 00$, B with 01, C with 10, and D with 11
- The state assignment can have a big impact on circuit complexity, but we won't worry about that too much in this class

Present State	Input	Next State	Output
A	0	A	0
A	1	B	0
B	0	C	0
B	1	B	0
C	0	D	0
C	1	B	0
D	0	A	0
D	1	B	1



Present State Q_1 Q_0	Input X	Next State Q_1 Q_0	Output Z
0 0	0	0 0	0
0 0	1	0 1	0
0 1	0	1 0	0
0 1	1	0 1	0
1 0	0	1 1	0
1 0	1	0 1	0
1 1	0	0 0	0
1 1	1	0 1	1

Step 3: Finding flip-flop input values

- Next we have to figure out how to actually make the flip-flops change from their present state into the desired next state
- This depends on what kind of flip-flops you use!
- We'll use two JKs. For each flip-flop Q_i , look at its present and next states, and determine what the inputs J_i and K_i should be in order to make that state change.

Present State		Input X	Next State		Flip flop inputs				Output Z
Q_1	Q_0		Q_1	Q_0	J_1	K_1	J_0	K_0	
0	0	0	0	0					0
0	0	1	0	1					0
0	1	0	1	0					0
0	1	1	0	1					0
1	0	0	1	1					0
1	0	1	0	1					0
1	1	0	0	0					0
1	1	1	0	1					1

Finding JK flip-flop input values

- For JK flip-flops, this is a little tricky. Recall the characteristic table:

J	K	Q(t+1)	Operation
0	0	Q(t)	No change
0	1	0	Reset
1	0	1	Set
1	1	Q'(t)	Complement

- If the present state of a JK flip-flop is 0 and we want the next state to be 1, then we have *two* choices for the JK inputs:
 - We can use JK= 10, to explicitly set the flip-flop's next state to 1
 - We can also use JK=11, to complement the current state 0
- So to change from 0 to 1, we must set J=1, but K could be *either* 0 or 1
- Similarly, the other possible state transitions can all be done in two different ways as well

JK excitation table

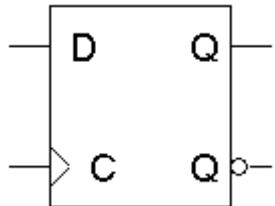
- An **excitation table** shows what flip-flop inputs are required in order to make a desired state change

$Q(t)$	$Q(t+1)$	J	K	Operation
0	0	0	x	No change/reset
0	1	1	x	Set/complement
1	0	x	1	Reset/complement
1	1	x	0	No change/set

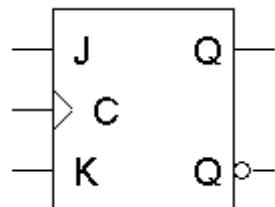
- This is the same information that's given in the characteristic table, but presented "backwards"

J	K	$Q(t+1)$	Operation
0	0	$Q(t)$	No change
0	1	0	Reset
1	0	1	Set
1	1	$Q'(t)$	Complement

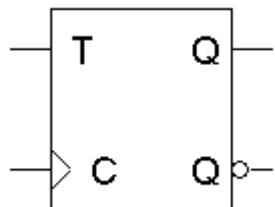
Excitation tables for all flip-flops



$Q(t)$	$Q(t+1)$	D	Operation
0	0	0	Reset
0	1	1	Set
1	0	0	Reset
1	1	1	Set



$Q(t)$	$Q(t+1)$	J	K	Operation
0	0	0	x	No change/reset
0	1	1	x	Set/complement
1	0	x	1	Reset/complement
1	1	x	0	No change/set



$Q(t)$	$Q(t+1)$	T	Operation
0	0	0	No change
0	1	1	Complement
1	0	1	Complement
1	1	0	No change

Back to the example

- Use the JK excitation table on the right to find the correct values for *each* flip-flop's inputs, based on its present and next states

$Q(t)$	$Q(t+1)$	J	K
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

Present State Q_1 Q_0	Input X	Next State Q_1 Q_0	Flip flop inputs				Output Z
			J_1	K_1	J_0	K_0	
0 0	0	0 0	0	x	0	x	0
0 0	1	0 1	0	x	1	x	0
0 1	0	1 0	1	x	x	1	0
0 1	1	0 1	0	x	x	0	0
1 0	0	1 1	x	0	1	x	0
1 0	1	0 1	x	1	1	x	0
1 1	0	0 0	x	1	x	1	0
1 1	1	0 1	x	1	x	0	1

Step 4: Find equations for the FF inputs and output

- Now you can make K-maps and find equations for each of the four flip-flop inputs, as well as for the output Z
- These equations are in terms of the present state and the inputs
- The advantage of using JK flip-flops is that there are many don't care conditions, which can result in simpler MSP equations

Present State		Input X	Next State		Flip flop inputs				Output Z
Q ₁	Q ₀		Q ₁	Q ₀	J ₁	K ₁	J ₀	K ₀	
0	0	0	0	0	0	x	0	x	0
0	0	1	0	1	0	x	1	x	0
0	1	0	1	0	1	x	x	1	0
0	1	1	0	1	0	x	x	0	0
1	0	0	1	1	x	0	1	x	0
1	0	1	0	1	x	1	1	x	0
1	1	0	0	0	x	1	x	1	0
1	1	1	0	1	x	1	x	0	1

$$\begin{aligned}J_1 &= X' Q_0 \\K_1 &= X + Q_0\end{aligned}$$

$$\begin{aligned}J_0 &= X + Q_1 \\K_0 &= X'\end{aligned}$$

$$Z = Q_1 Q_0 X$$

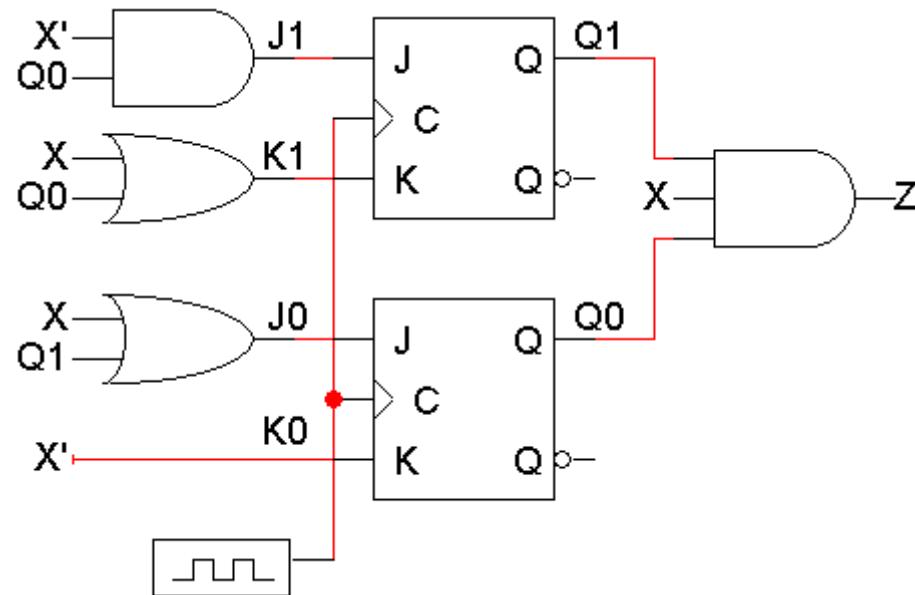
Step 5: Build the circuit

- Lastly, we use these simplified equations to build the completed circuit

$$J_1 = X' Q_0$$
$$K_1 = X + Q_0$$

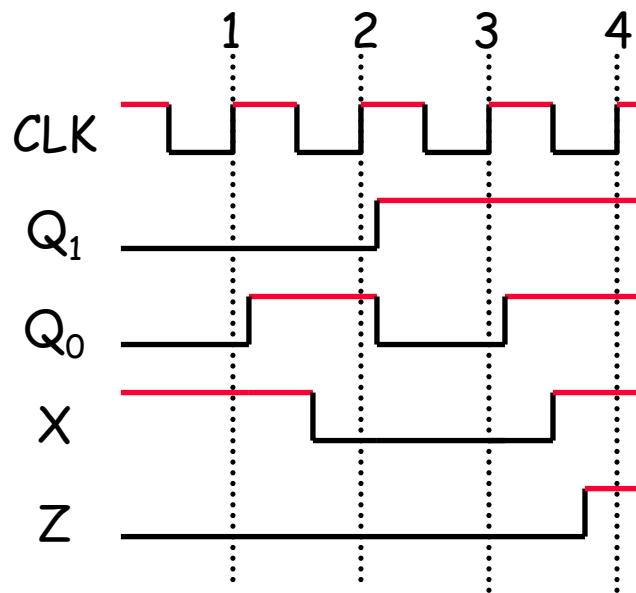
$$J_0 = X + Q_1$$
$$K_0 = X'$$

$$Z = Q_1 Q_0 X$$



Timing diagram

- Here is one example timing diagram for our sequence detector
 - The flip-flops Q_1Q_0 start in the initial state, 00
 - On the first three positive clock edges, X is 1, 0, and 0. These inputs cause Q_1Q_0 to change, so after the third edge $Q_1Q_0 = 11$
 - Then when $X=1$, Z becomes 1 also, meaning that 1001 was found
- The output Z does not have to change at positive clock edges. Instead, it may change whenever X changes, since $Z = Q_1Q_0X$



Building the same circuit with D flip-flops

- What if you want to build the circuit using D flip-flops instead?
- We already have the state table and state assignments, so we can just start from Step 3, finding the flip-flop input values
- D flip-flops have only one input, so our table only needs two columns for D_1 and D_0

Present State Q_1 Q_0		Input X	Next State Q_1 Q_0		Flip-flop inputs D_1 D_0		Output Z
0	0	0	0	0			0
0	0	1	0	1			0
0	1	0	1	0			0
0	1	1	0	1			0
1	0	0	1	1			0
1	0	1	0	1			0
1	1	0	0	0			0
1	1	1	0	1			1

D flip-flop input values (Step 3)

- The D excitation table is pretty boring; set the D input to whatever the next state should be
- You don't even need to show separate columns for D_1 and D_0 ; you can just use the Next State columns

$Q(t)$	$Q(t+1)$	D	Operation
0	0	0	Reset
0	1	1	Set
1	0	0	Reset
1	1	1	Set

Present State		Input X	Next State		Flip flop inputs		Output Z
Q_1	Q_0		Q_1	Q_0	D_1	D_0	
0	0	0	0	0	0	0	0
0	0	1	0	1	0	1	0
0	1	0	1	0	1	0	0
0	1	1	0	1	0	1	0
1	0	0	1	1	1	1	0
1	0	1	0	1	0	1	0
1	1	0	0	0	0	0	0
1	1	1	0	1	0	1	1

Finding equations (Step 4)

- You can do K-maps again, to find:

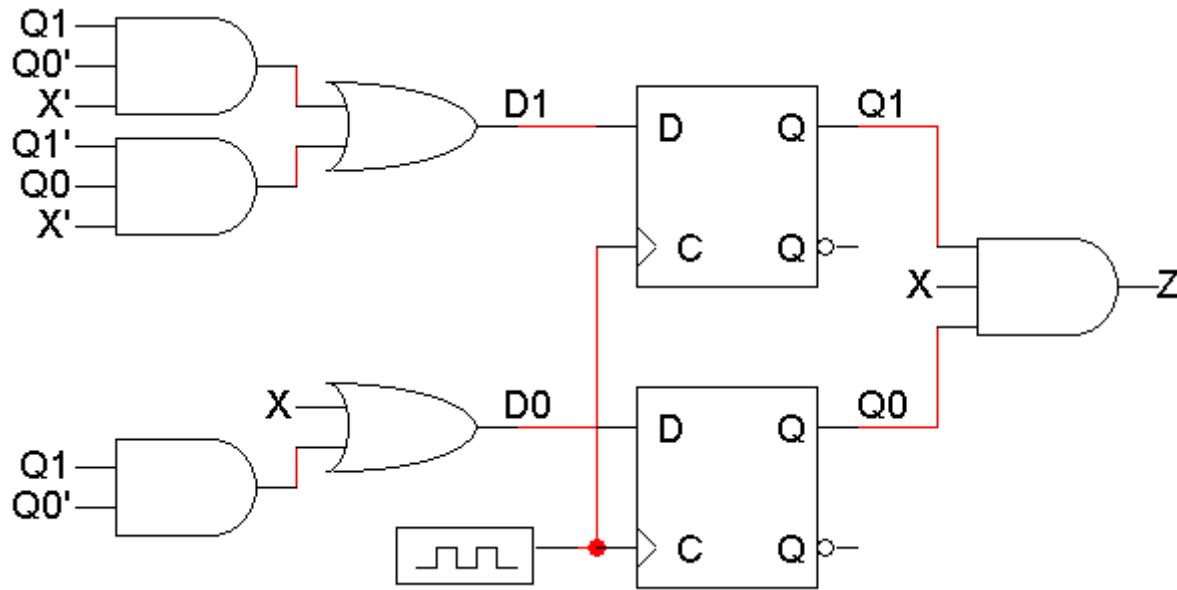
$$D_1 = Q_1 Q_0' X' + Q_1' Q_0 X'$$

$$D_0 = X + Q_1 Q_0'$$

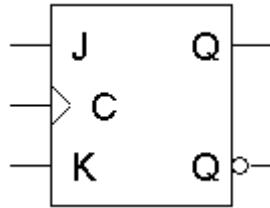
$$Z = Q_1 Q_0 X$$

Present State $Q_1 \quad Q_0$		Input X	Next State $Q_1 \quad Q_0$		Flip flop inputs $D_1 \quad D_0$		Output Z
0	0	0	0	0	0	0	0
0	0	1	0	1	0	1	0
0	1	0	1	0	1	0	0
0	1	1	0	1	0	1	0
1	0	0	1	1	1	1	0
1	0	1	0	1	0	1	0
1	1	0	0	0	0	0	0
1	1	1	0	1	0	1	1

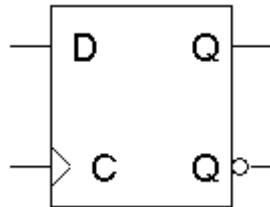
Building the circuit (Step 5)



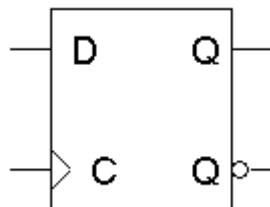
Flip-flop comparison



JK flip-flops are good because there are many don't care values in the flip-flop inputs, which can lead to a simpler circuit



D flip-flops have the advantage that you don't have to set up flip-flop inputs at all, since $Q(t+1) = D$. However, the D input equations are usually more complex than JK input equations



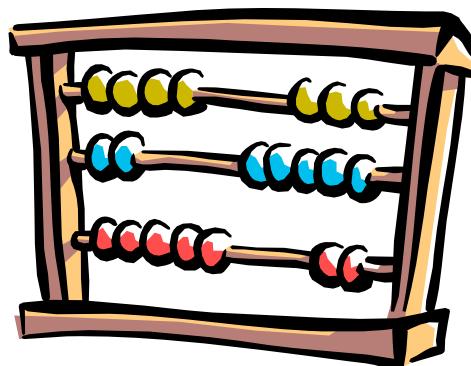
In practice, D flip-flops are used more often

- There is only one input for each flip-flop, not two
- There are no excitation tables to worry about
- D flip-flops can be implemented with slightly less hardware than JK flip-flops

Summary

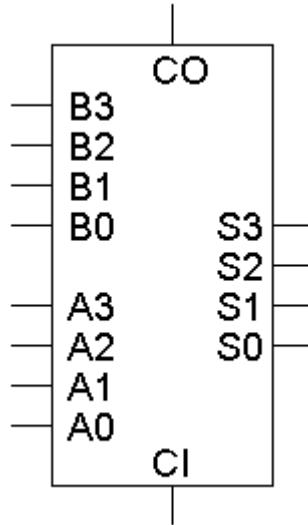
- The basic sequential circuit design procedure:
 - Make a state table and, if desired, a state diagram. This step is usually the hardest
 - Assign binary codes to the states if you didn't already
 - Use the present states, next states, and flip-flop excitation tables to find the flip-flop input values
 - Write simplified equations for the flip-flop inputs and outputs and build the circuit

Arithmetic-Logic Units (ALUs)



The four-bit adder

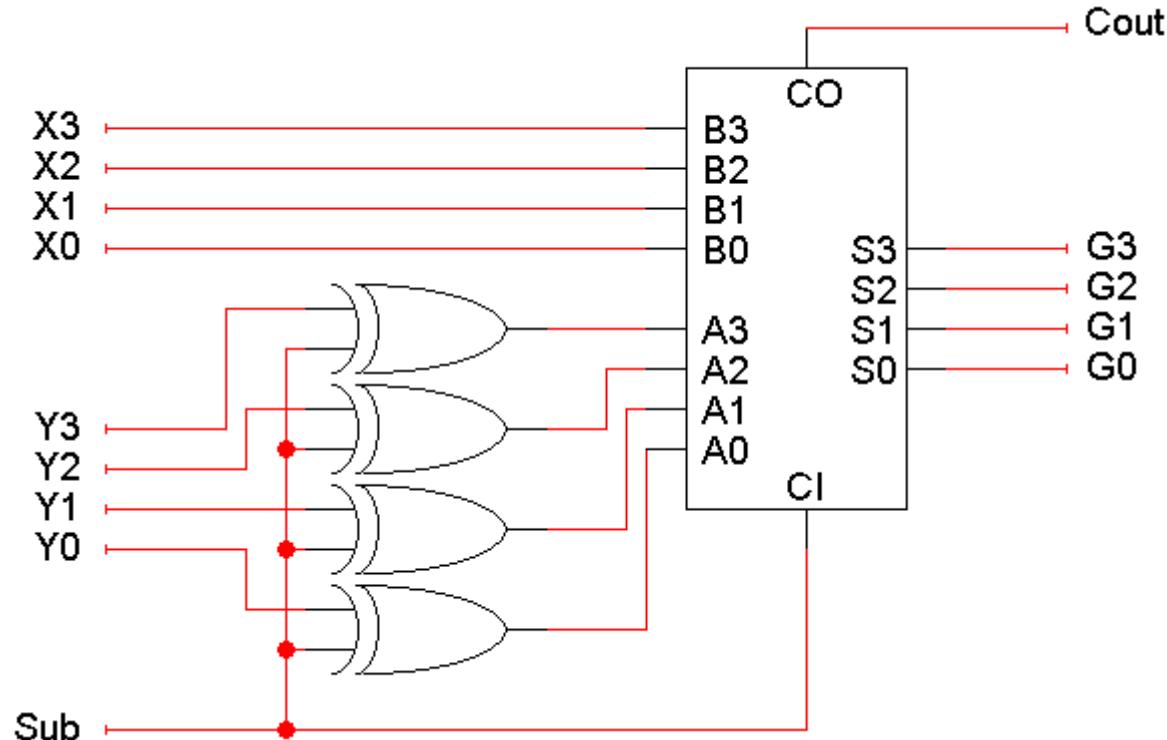
- The basic four-bit adder *always* computes $S = A + B + CI$



- But by changing what goes into the adder inputs A, B and CI, we can change the adder output S
- This is also what we did to build the combined adder-subtractor circuit

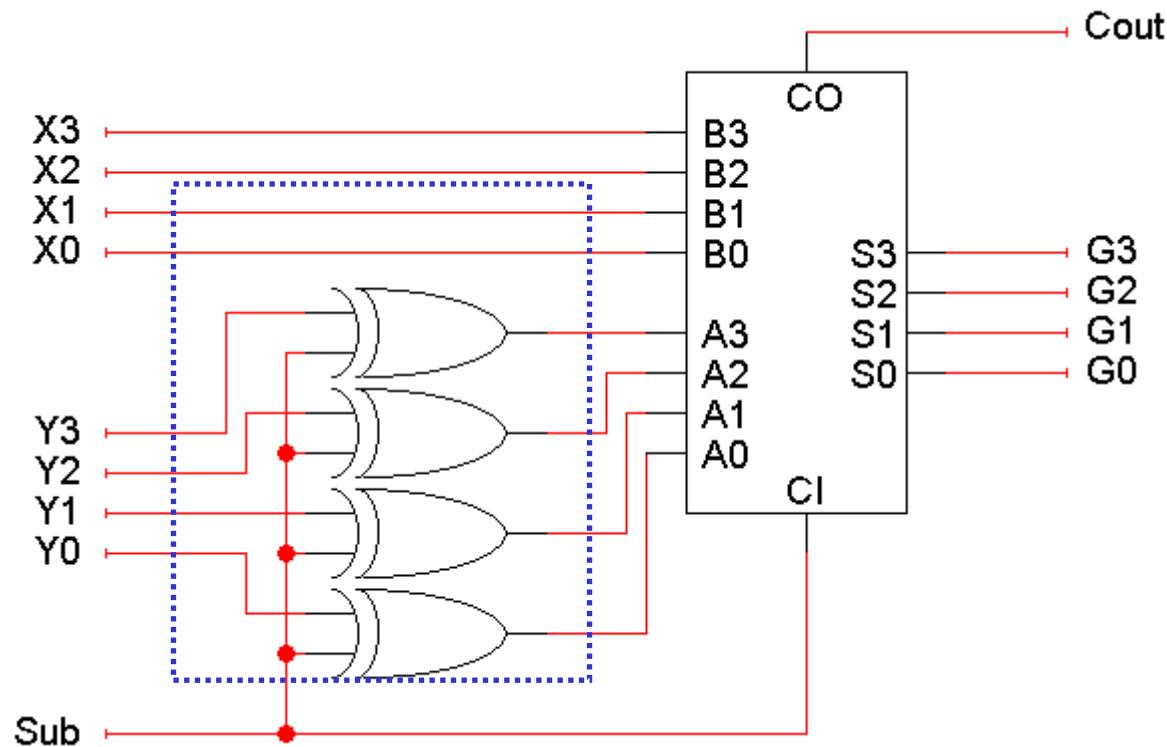
It's the adder-subtractor again!

- Here the signal Sub and some XOR gates alter the adder inputs
 - When $\text{Sub} = 0$, the adder inputs A, B, CI are Y, X, 0, so the adder produces $G = X + Y + 0$, or just $X + Y$
 - When $\text{Sub} = 1$, the adder inputs are Y' , X and 1, so the adder output is $G = X + Y' + 1$, or the two's complement operation $X - Y$



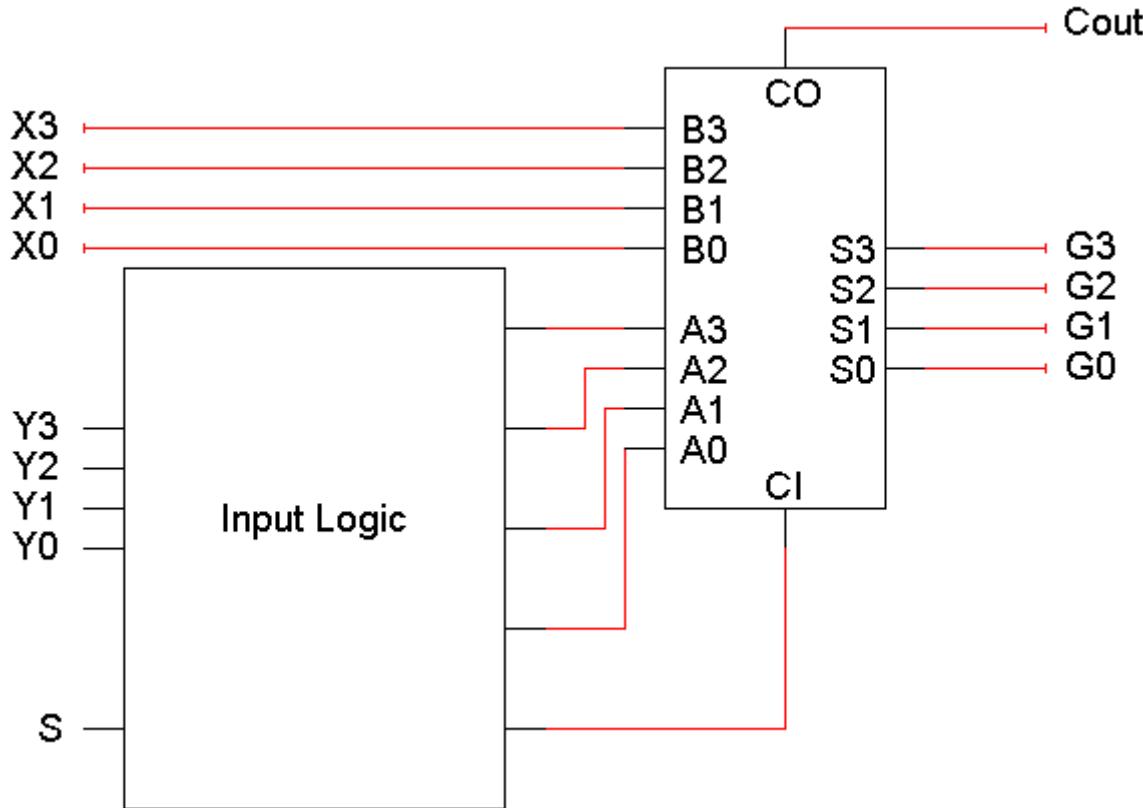
The multi-talented adder

- So we have one adder performing two separate functions
- "Sub" acts like a function select input which determines whether the circuit performs addition or subtraction
- Circuit-wise, all "Sub" does is modify the adder's inputs A and CI



Modifying the adder inputs

- By following the same approach, we can use an adder to compute *other* functions as well
- We just have to figure out which functions we want, and then put the right circuitry into the "Input Logic" box



Some more possible functions

- We already saw how to set adder inputs A , B and CI to compute either $X + Y$ or $X - Y$
- How can we produce the increment function $G = X + 1$?

One way: Set $A = 0000$, $B = X$, and $CI = 1$

- How about decrement: $G = X - 1$?

$A = 1111 (-1)$, $B = X$, $CI = 0$

- How about transfer: $G = X$?

$A = 0000$, $B = X$, $CI = 0$

This is almost the same as the increment function!

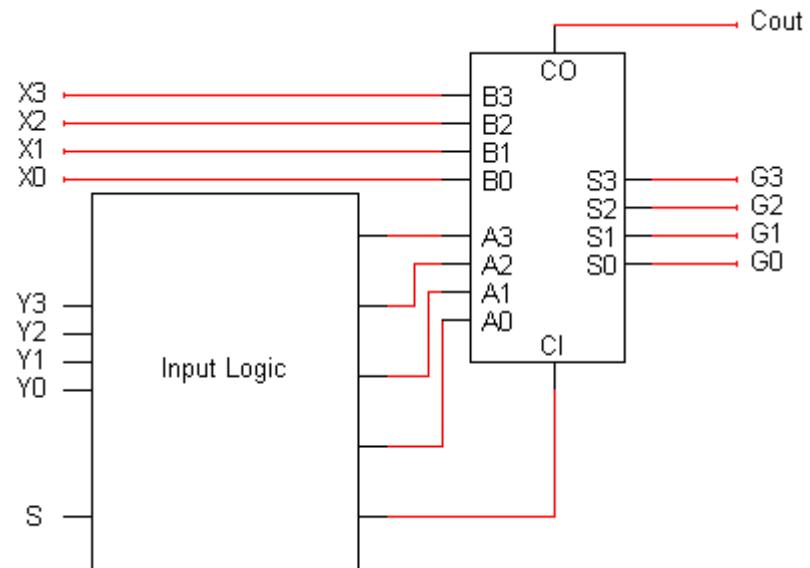


Table of arithmetic functions

- Here are some of the different possible arithmetic operations
- We'll need some way to specify which function we're interested in, so we've randomly assigned a selection code to each operation

S_2	S_1	S_0	Arithmetic operation	
0	0	0	X	(transfer)
0	0	1	$X + 1$	(increment)
0	1	0	$X + Y$	(add)
0	1	1	$X + Y + 1$	
1	0	0	$X + Y'$	(1C subtraction)
1	0	1	$X + Y' + 1$	(2C subtraction)
1	1	0	$X - 1$	(decrement)
1	1	1	X	(transfer)

Mapping the table to an adder

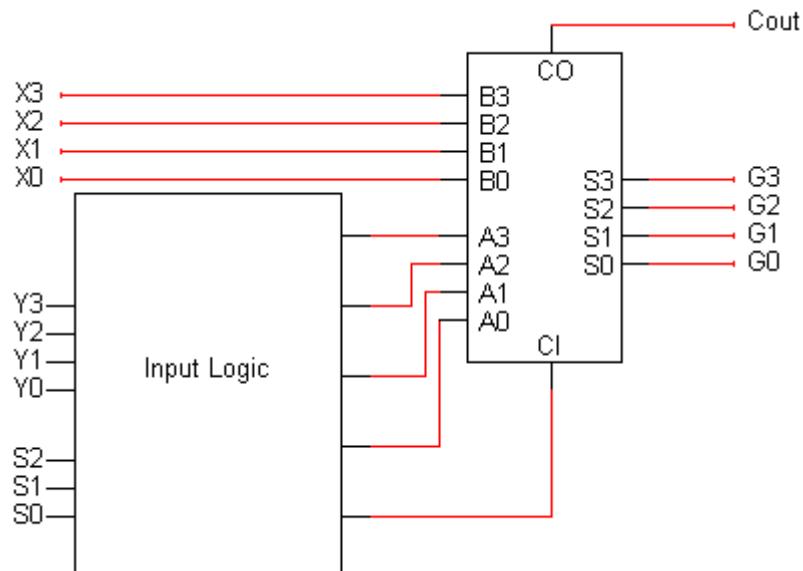
- This second table shows what the adder's inputs should be for each of our eight desired arithmetic operations
 - Adder input CI is always the same as selection code bit S_0
 - B is always set to X
 - A depends only on S_2 and S_1
- These equations depend on both the desired operations and the assignment of selection codes

Selection code			Desired arithmetic operation $G(A + B + CI)$		Required adder inputs		
S_2	S_1	S_0	A	B	CI		
0	0	0	X	(transfer)	0000	X	0
0	0	1	$X + 1$	(increment)	0000	X	1
0	1	0	$X + Y$	(add)	Y	X	0
0	1	1	$X + Y + 1$		Y	X	1
1	0	0	$X + Y'$	(1C subtraction)	Y'	X	0
1	0	1	$X + Y' + 1$	(2C subtraction)	Y'	X	1
1	1	0	$X - 1$	(decrement)	1111	X	0
1	1	1	X	(transfer)	1111	X	1

Building the input logic

- All we need to do is compute the adder input A , given the arithmetic unit input Y and the function select code S (actually just S_2 and S_1)
- Here is an abbreviated truth table:

S_2	S_1	A
0	0	0000
0	1	Y
1	0	Y'
1	1	1111

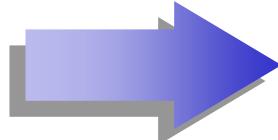


- We want to pick one of these four possible values for A , depending on S_2 and S_1

Primitive gate-based input logic

- We could build this circuit using primitive gates
- If we want to use K-maps for simplification, then we should first expand out the abbreviated truth table
 - The Y that appears in the output column (A) is actually an input
 - We make that explicit in the table on the right
- Remember A and Y are each 4 bits long!

S_2	S_1	A
0	0	0000
0	1	y
1	0	y'
1	1	1111



S_2	S_1	Y_i	A_i
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

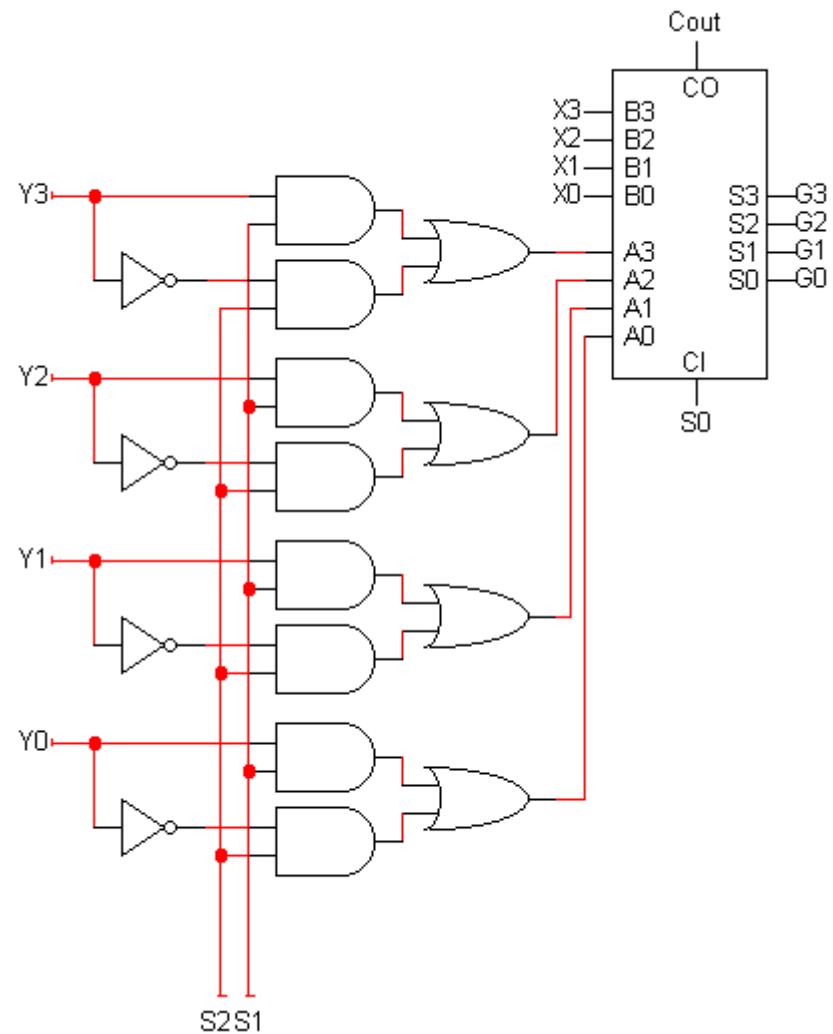
Primitive gate implementation

- From the truth table, we can find an MSP:

		S_1	
	0	0	1
S_2	0	1	0
		y_i	

$$A_i = S_2 y'_i + S_1 y_i$$

- Again, we have to repeat this once for each bit y_3-y_0 , connecting to the adder inputs A_3-A_0
- This completes our arithmetic unit



Bitwise operations

- Most computers also support logical operations like AND, OR and NOT, but extended to multi-bit **words** instead of just single bits
- To apply a logical operation to two words X and Y , apply the operation on each pair of bits X_i and Y_i :

$$\begin{array}{r} 1 \textcolor{magenta}{0} \textcolor{green}{1} \textcolor{red}{1} \\ \text{AND } 1 \textcolor{magenta}{1} \textcolor{blue}{1} \textcolor{red}{0} \\ \hline 1 \textcolor{magenta}{0} \textcolor{red}{1} \textcolor{green}{0} \end{array}$$

$$\begin{array}{r} 1 \textcolor{magenta}{0} \textcolor{green}{1} \textcolor{red}{1} \\ \text{OR } 1 \textcolor{blue}{1} \textcolor{magenta}{1} \textcolor{red}{0} \\ \hline 1 \textcolor{blue}{1} \textcolor{magenta}{1} \textcolor{red}{1} \end{array}$$

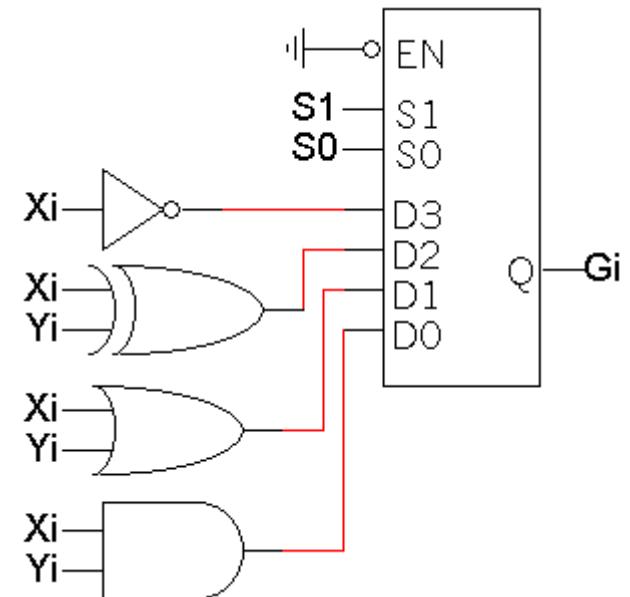
$$\begin{array}{r} 1 \textcolor{magenta}{0} \textcolor{green}{1} \textcolor{red}{1} \\ \text{XOR } 1 \textcolor{blue}{1} \textcolor{magenta}{1} \textcolor{red}{0} \\ \hline 0 \textcolor{blue}{1} \textcolor{green}{0} \textcolor{red}{1} \end{array}$$

- We've already seen this informally in two's-complement arithmetic, when we talked about "complementing" all the bits in a number

Defining a logic unit

- A logic unit supports different logical functions on two multi-bit inputs X and Y , producing an output G
- This abbreviated table shows four possible functions and assigns a selection code S to each

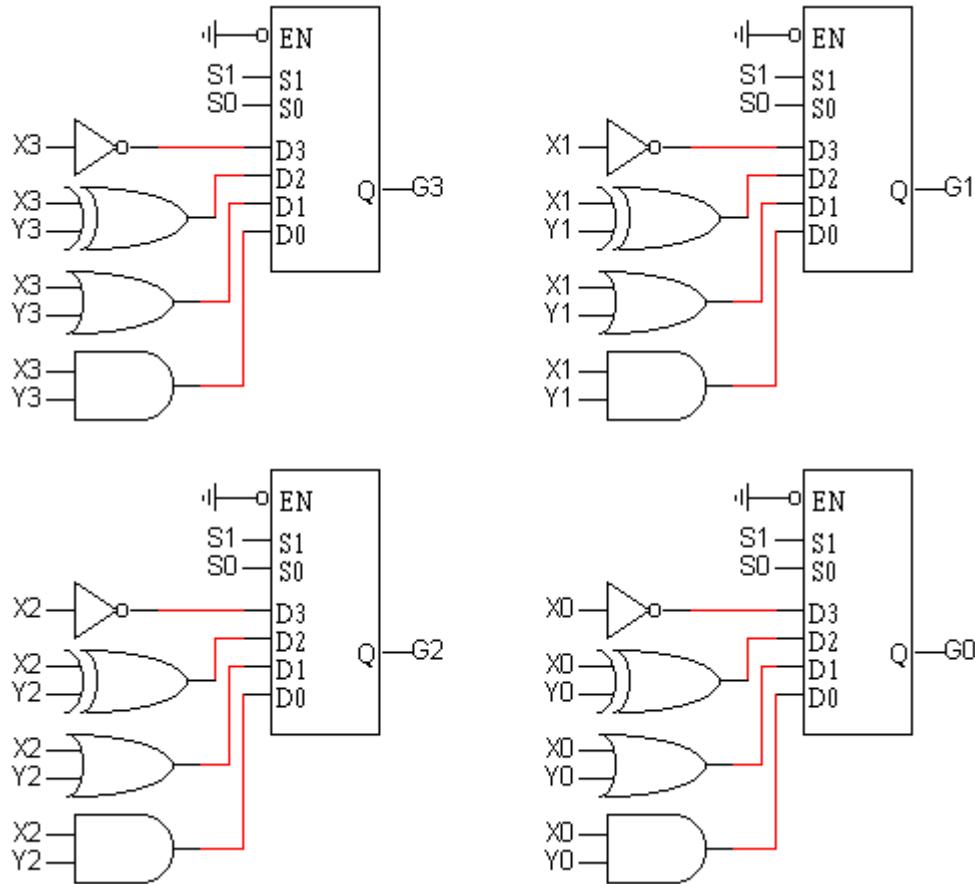
S_1	S_0	Output
0	0	$G_i = X_i Y_i$
0	1	$G_i = X_i + Y_i$
1	0	$G_i = X_i \oplus Y_i$
1	1	$G_i = X_i'$



- We'll just use multiplexers and some primitive gates to implement this
- Again, we need one multiplexer for each bit of X and Y

Our simple logic unit

- **Inputs:**
 - X (4 bits)
 - Y (4 bits)
 - S (2 bits)
- **Outputs:**
 - G (4 bits)



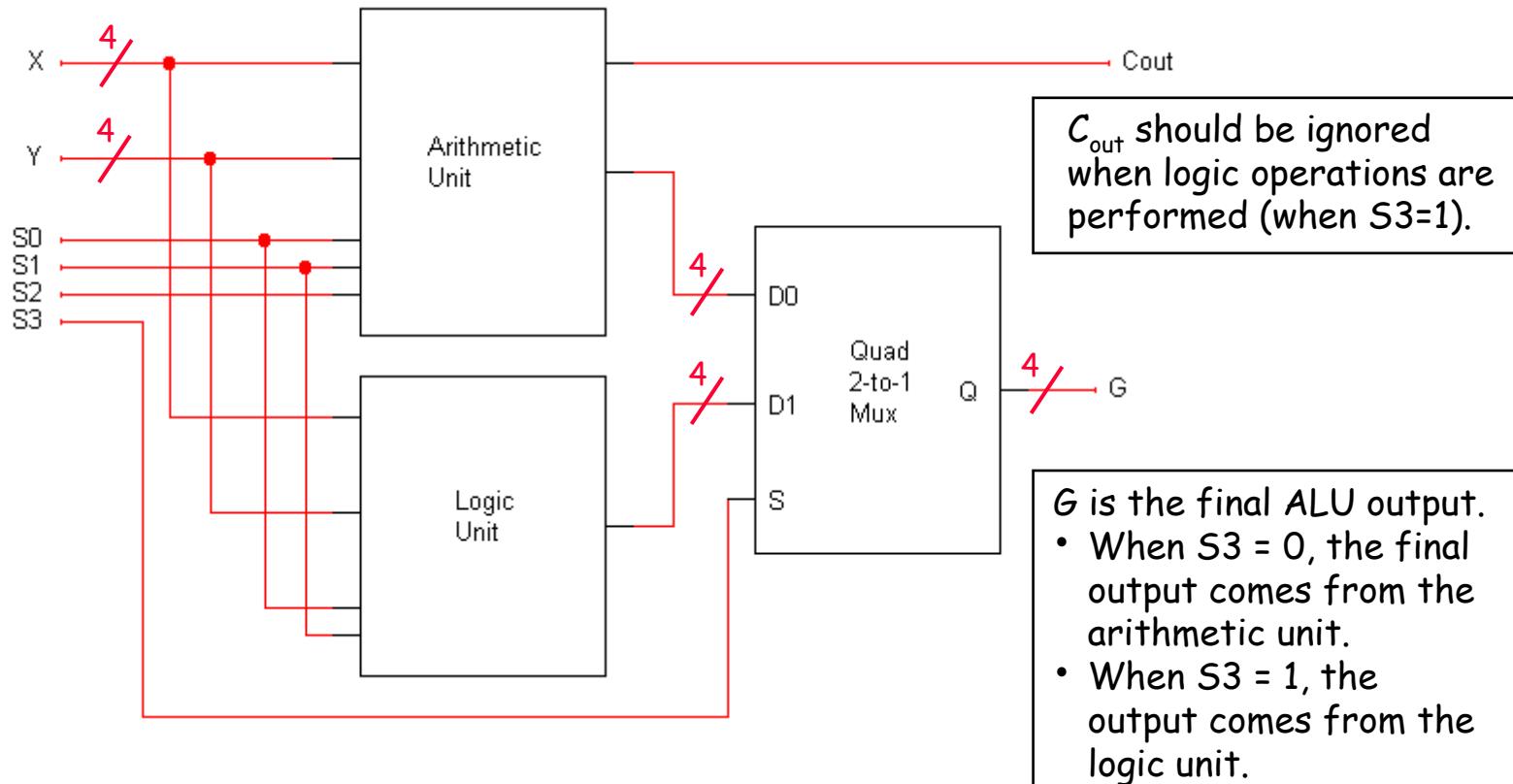
Our ALU function table

- This table shows a sample function table for an ALU
- All of the arithmetic operations have $S_3=0$, and all of the logical operations have $S_3=1$
- These are the same functions we saw when we built our arithmetic and logic units a few minutes ago
- Since our ALU only has 4 logical operations, we don't need S_2 . The operation done by the logic unit depends only on S_1 and S_0

S_3	S_2	S_1	S_0	Operation
0	0	0	0	$G = X$
0	0	0	1	$G = X + 1$
0	0	1	0	$G = X + Y$
0	0	1	1	$G = X + Y + 1$
0	1	0	0	$G = X + Y'$
0	1	0	1	$G = X + Y' + 1$
0	1	1	0	$G = X - 1$
0	1	1	1	$G = X$
1	x	0	0	$G = X \text{ and } Y$
1	x	0	1	$G = X \text{ or } Y$
1	x	1	0	$G = X \oplus Y$
1	x	1	1	$G = X'$

A complete ALU circuit

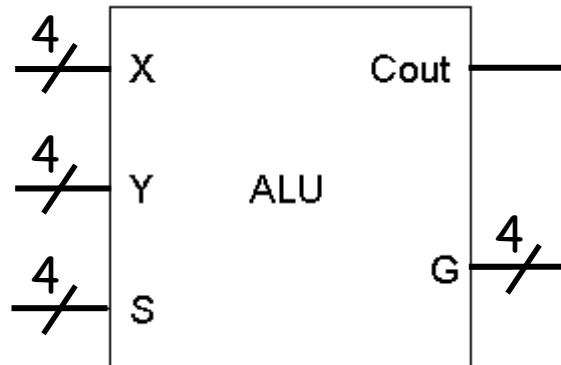
The / and 4 on a line indicate that it's actually four lines.



The arithmetic and logic units share the select inputs S_1 and S_0 , but only the arithmetic unit uses S_2 .

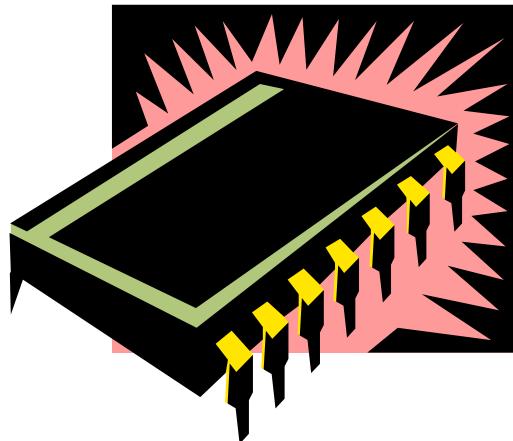
The completed ALU

- This ALU is a good example of hierarchical design
 - With the 12 inputs, the truth table would have had $2^{12} = 4096$ lines. That's an awful lot of paper.
 - Instead, we were able to use components that we've seen before to construct the entire circuit from a couple of easy-to-understand components
- As always, we encapsulate the complete circuit in a "black box" so we can reuse it in fancier circuits.



Memory

- Sequential circuits all depend upon the presence of memory.
 - A flip-flop can store one bit of information.
 - A register can store a single "word," typically 32-64 bits.
- **Memory** allows us to store even larger amounts of data.
 - Random Access Memory (RAM - volatile)
 - Static RAM (SRAM)
 - Dynamic RAM (DRAM)
 - Read Only Memory (ROM - nonvolatile)



Introduction to RAM

- Random-access memory, or RAM, provides large quantities of temporary storage in a computer system.
 - Memory cells can be accessed to transfer information to or from any desired location, with the access taking the same time regardless of the location
- A RAM can store *many* values.
 - An address will specify which memory value we're interested in.
 - Each value can be a multiple-bit word (e.g., 32 bits).
- We'll refine the memory properties as follows:

A RAM should be able to:

- Store many words, one per address
- Read the word that was saved at a particular address
- Change the word that's saved at a particular address

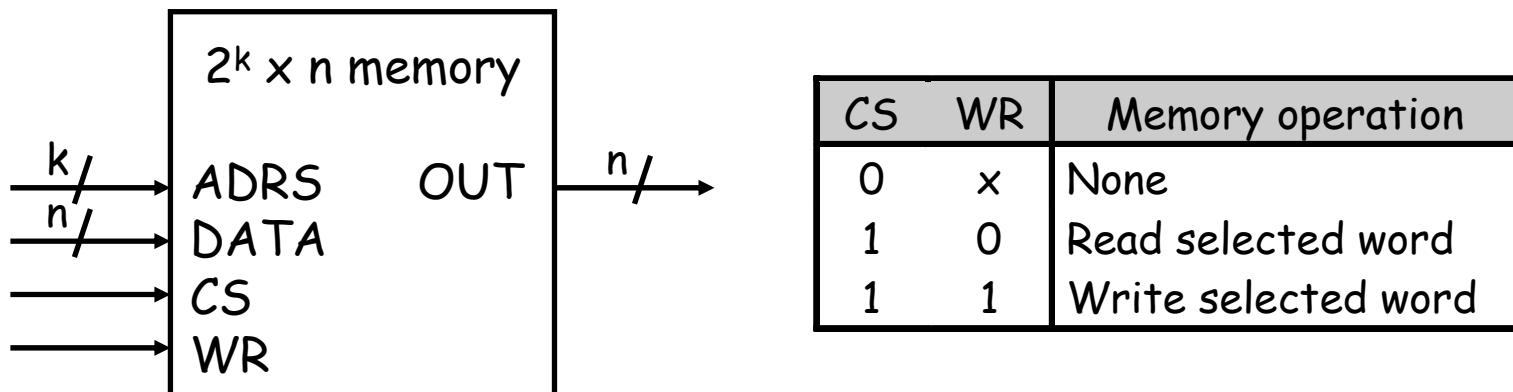
Picture of memory

- You can think of computer memory as being one big array of data.
 - The address serves as an array index.
 - Each address refers to one word of data.
- You can read or modify the data at any given memory address, just like you can read or modify the contents of an array at any given index.

Address	Data
00000000	
00000001	
00000002	
.	
.	
.	
.	
.	
.	
.	
.	
.	
.	
.	
FFFFFFFFFFD	
FFFFFFFFFFE	
FFFFFFFFFF	

 Word₃

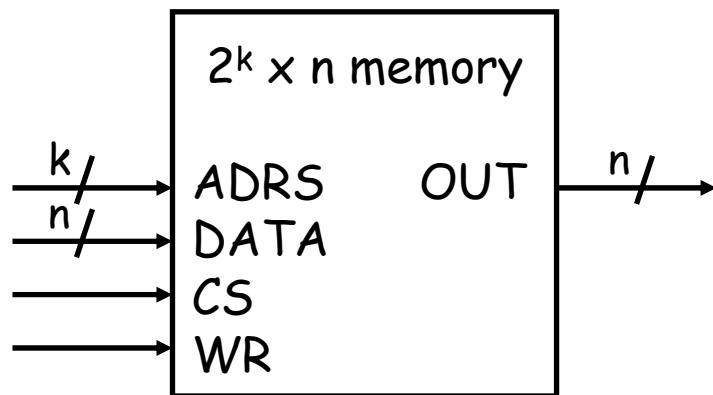
Block diagram of RAM



- This block diagram introduces the main interface to RAM.
 - A Chip Select, **CS**, enables or disables the RAM.
 - **ADRS** specifies the address or location to read from or write to.
 - **WR** selects between reading from or writing to the memory.
 - ▶ To read from memory, WR should be set to 0.
OUT will be the n-bit value stored at ADRS.
 - ▶ To write to memory, we set WR = 1.
DATA is the n-bit value to save in memory.
- This interface makes it easy to combine RAMs together, as we'll see.

Memory sizes

- We refer to this as a $2^k \times n$ memory.
 - There are k *address lines*, which can specify one of 2^k addresses.
 - Each address contains an n -bit word.



- For example, a $2^{24} \times 16$ RAM contains $2^{24} = 16M$ words, each 16 bits long.
 - The RAM would need 24 address lines.
 - The total *storage capacity* is $2^{24} \times 16 = 2^{28}$ bits.

Size matters!

- Memory sizes are usually specified in numbers of **bytes** (1 byte= 8 bits).
- The 2^{28} -bit memory on the previous page translates into:
$$2^{28} \text{ bits} / 8 \text{ bits per byte} = 2^{25} \text{ bytes}$$
- With the abbreviations below, this is equivalent to 32 megabytes.

	Prefix	Base 2	Base 10
K	Kilo	$2^{10} = 1,024$	$10^3 = 1,000$
M	Mega	$2^{20} = 1,048,576$	$10^6 = 1,000,000$
G	Giga	$2^{30} = 1,073,741,824$	$10^9 = 1,000,000,000$

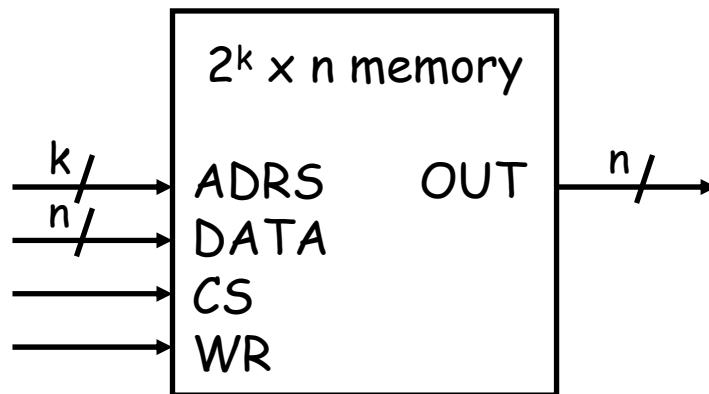
Typical memory sizes

Address	Data
00000000	
00000001	
00000002	
.	
.	
.	
.	
.	
.	
.	
.	
.	
.	
FFFFFFFFFFD	
FFFFFFFFFFE	
FFFFFFFFFFF	

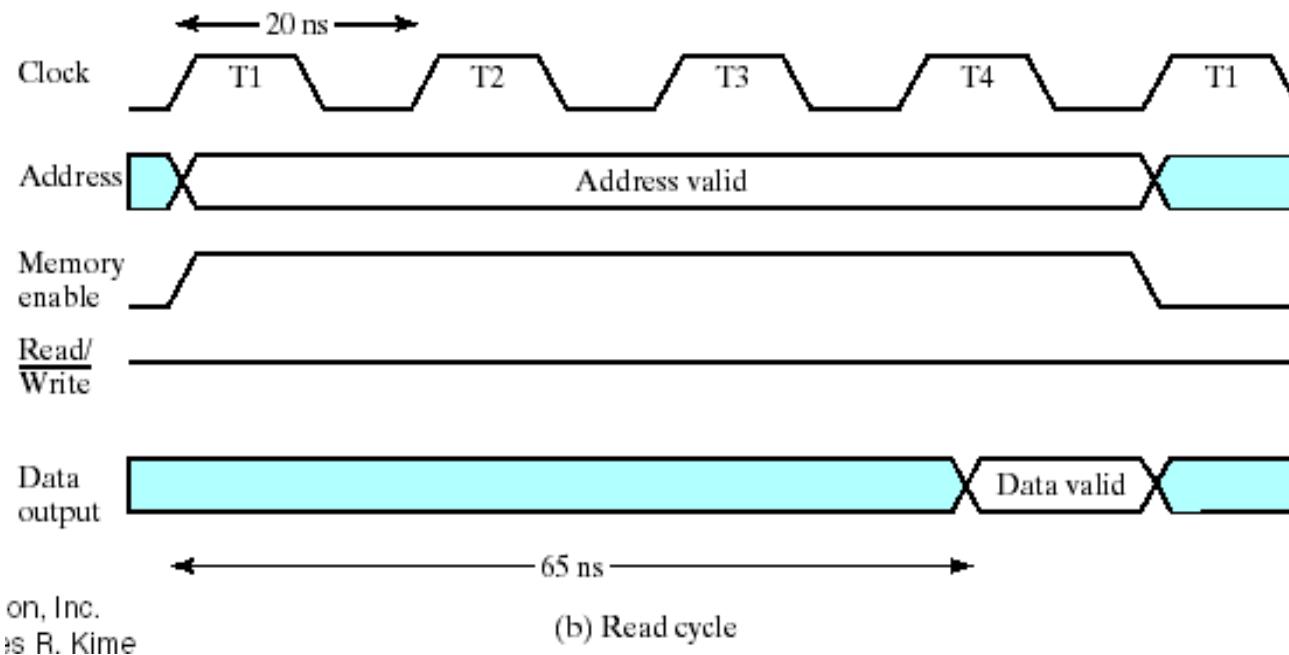
- Some typical memory capacities:
 - PCs usually come with 1-3GB RAM.
 - PDAs have 64-256MB of memory.
 - Digital cameras and MP3 players can have 32MB or more of storage.
- Many operating systems implement **virtual memory**, which makes the memory seem larger than it really is.
 - Most systems allow up to 32-bit addresses. This works out to 2^{32} , or about four billion, different possible addresses.
 - With a data size of one byte, the result is apparently a 4GB memory!
 - The operating system uses hard disk space as a substitute for "real" memory.

Reading RAM

- To *read* from this RAM, the controlling circuit must:
 - Enable the chip by ensuring $CS = 1$.
 - Select the read operation, by setting $WR = 0$.
 - Send the desired address to the $ADRS$ input.
 - The contents of that address appear on OUT after a little while.
- Notice that the $DATA$ input is unused for read operations.



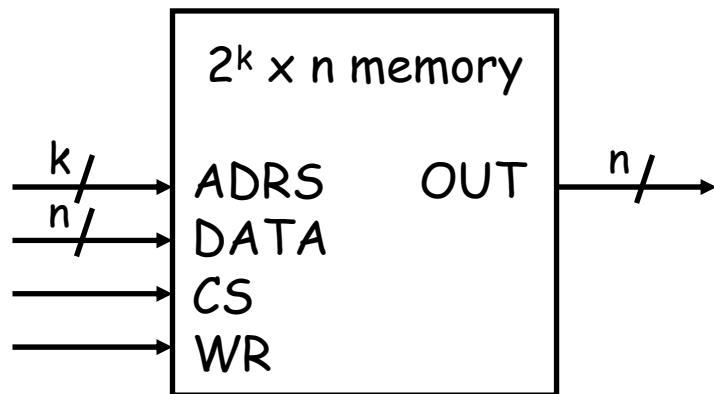
Reading RAM



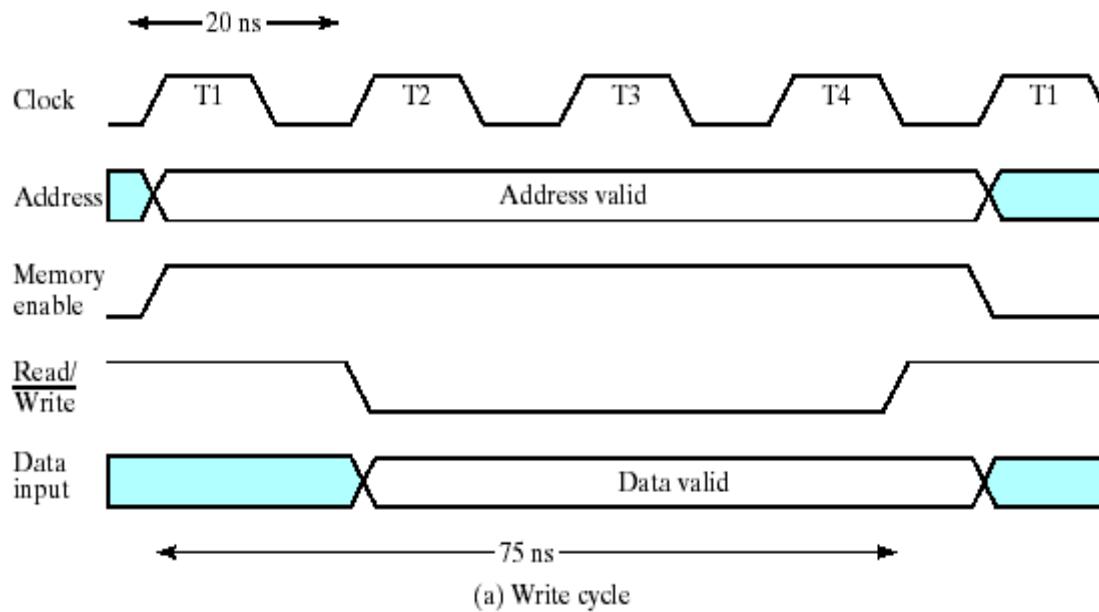
- 50 MHz CPU - 20 ns clock cycle time
- **Memory access time** = 65 ns
 - Maximum time from the application of the address to the appearance of the data at the Data Output

Writing RAM

- To *write* to this RAM, you need to:
 - Enable the chip by setting CS = 1.
 - Select the write operation, by setting WR = 1.
 - Send the desired address to the ADRS input.
 - Send the word to store to the DATA input.
- The output OUT is not needed for memory write operations.



Writing RAM



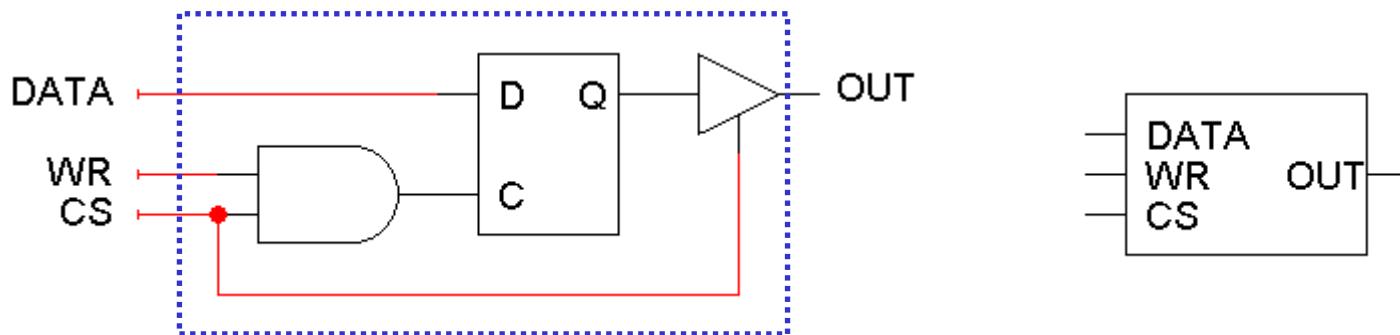
- 50 MHz CPU - 20 ns clock cycle time
- **Write cycle time = 75 ns**
 - Maximum time from the application of the address to the completion of all internal memory operations to store a word

Static memory

- How can you implement the memory chip?
- There are many different kinds of RAM.
 - We'll start off discussing **static memory**, which is most commonly used in caches and video cards.
 - Later we mention a little about **dynamic memory**, which forms the bulk of a computer's main memory.
- Static memory is modeled using one *latch* for each bit of storage.
- Why use latches instead of flip flops?
 - A latch can be made with only two NAND or two NOR gates, but a flip-flop requires at least twice that much hardware.
 - In general, smaller is faster, cheaper and requires less power.
 - The tradeoff is that getting the timing exactly right is a pain.

Starting with latches

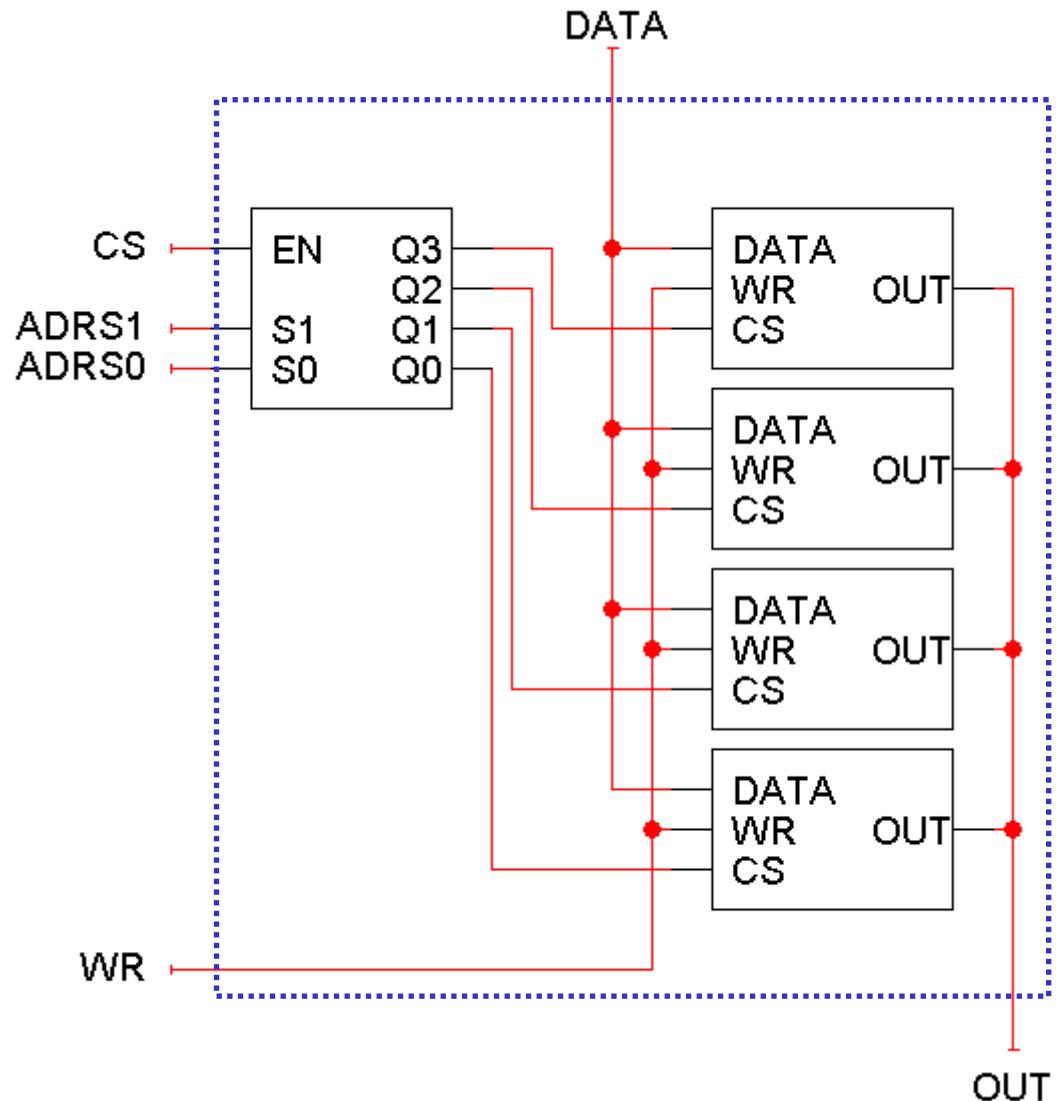
- To start, we can use one latch to store each bit. A one-bit RAM cell is shown here.



- Since this is just a one-bit memory, an ADRS input is not needed.
- Writing to the RAM cell:
 - When $CS = 1$ and $WR = 1$, the latch control input will be 1.
 - The DATA input is thus saved in the D latch.
- Reading from the RAM cell and maintaining the current contents:
 - When $CS = 0$ or when $WR = 0$, the latch control input is also 0, so the latch just maintains its present state.
 - The current latch content will appear on OUT when $CS = 1$.

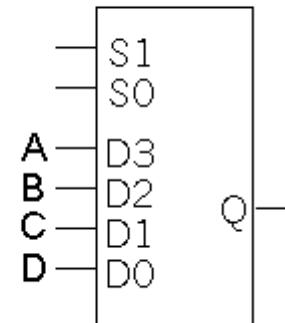
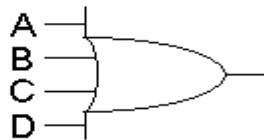
My first RAM

- We can use these cells to make a 4×1 RAM.
- Since there are four words, ADRS is two bits.
- Each word is only one bit, so DATA and OUT are one bit each.
- Word selection is done with a decoder attached to the CS inputs of the RAM cells. Only one cell can be read or written at a time.
- Notice that the outputs are connected together with a *single* line!



Connecting outputs together

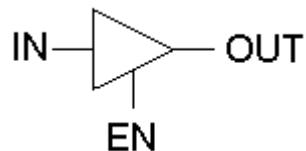
- In normal practice, it's bad to connect outputs together. If the outputs have different values, then a conflict arises.
- The standard way to "combine" outputs is to use OR gates or muxes.



- This can get expensive, with many wires and gates with large fan-ins.

Those funny triangles

- The triangle represents a **three-state buffer**.
- Unlike regular logic gates, the output can be one of *three* different possibilities, as shown in the table.

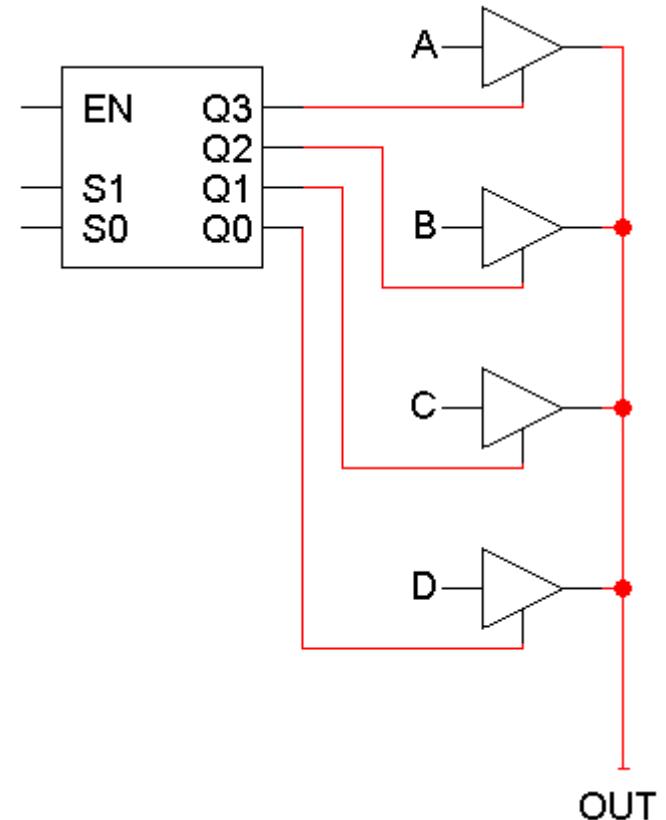


EN	IN	OUT
0	x	Disconnected
1	0	0
1	1	1

- “Disconnected” means no output appears at all, in which case it’s safe to connect OUT to another output signal.
- The disconnected value is also sometimes called **high impedance** or **Hi-Z**.

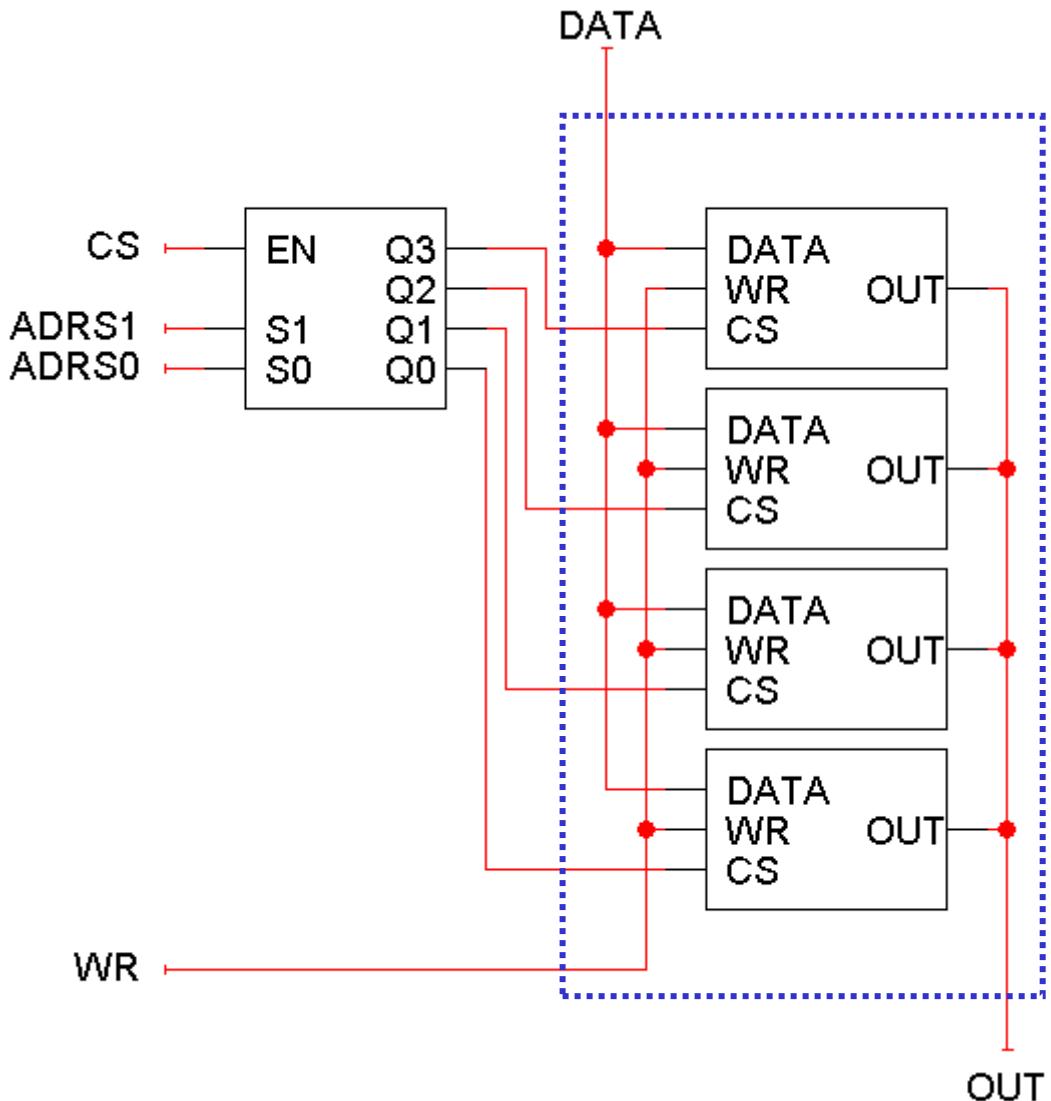
Connecting three-state buffers together

- You can connect several three-state buffer outputs together if you can *guarantee* that only one of them is enabled at any time.
- The easiest way to do this is to use a decoder!
- If the decoder is disabled, then all the three-state buffers will appear to be disconnected, and OUT will also appear disconnected.
- If the decoder is enabled, then exactly one of its outputs will be true, so only one of the tri-state buffers will be connected and produce an output.
- The net result is we can save some wire and gate costs. We also get a little more flexibility in putting circuits together.



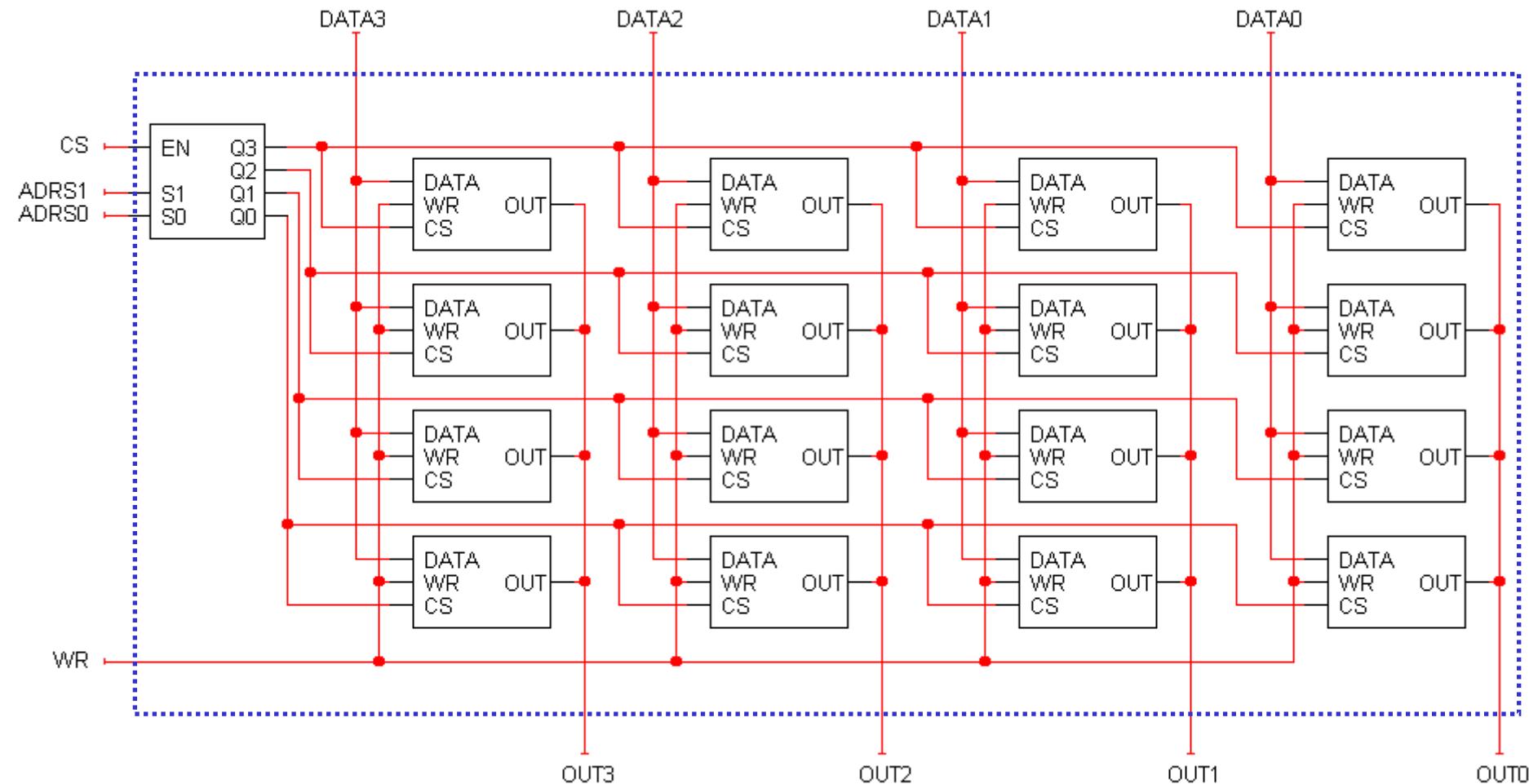
Bigger and better

- Here is the 4×1 RAM once again.
- How can we make a “wider” memory with more bits per word, like maybe a 4×4 RAM?
- Duplicate the stuff in the blue box!

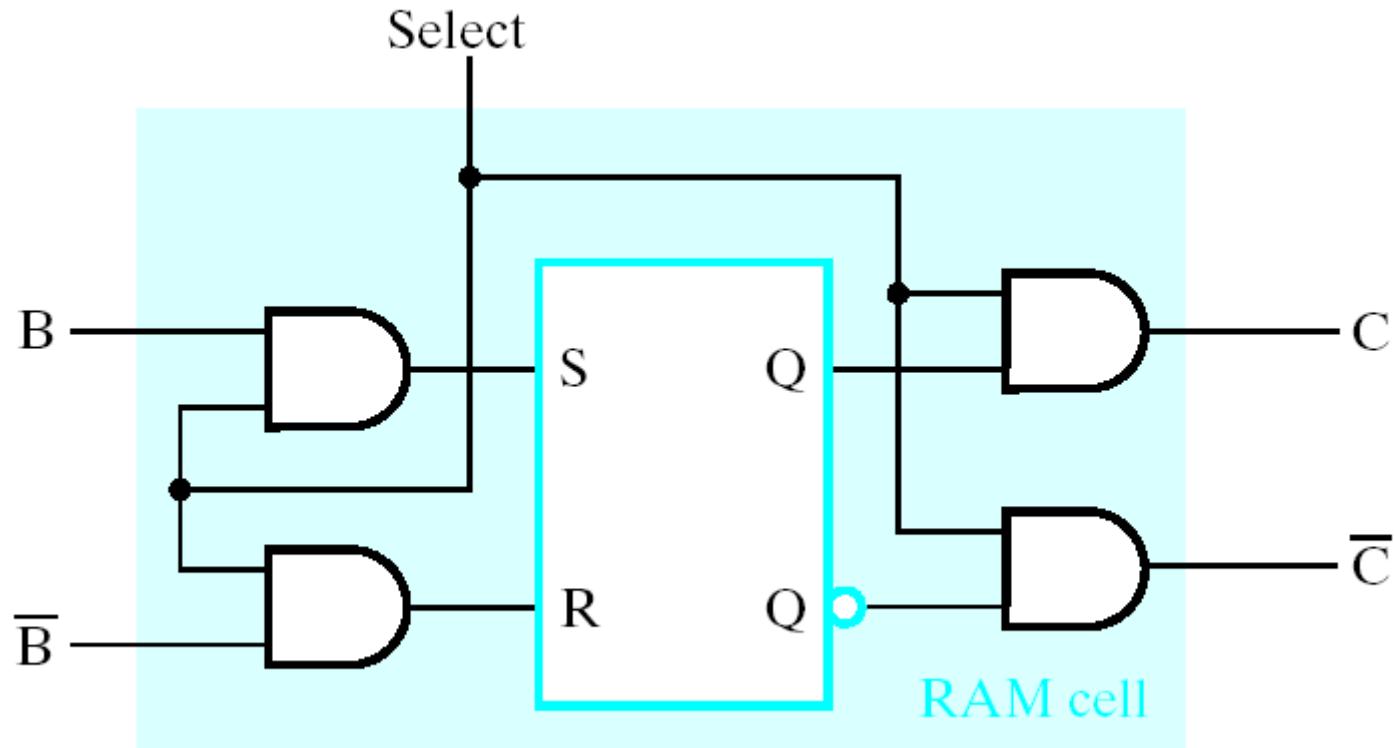


A 4×4 RAM

- DATA and OUT are now each *four* bits long, so you can read and write four-bit words.

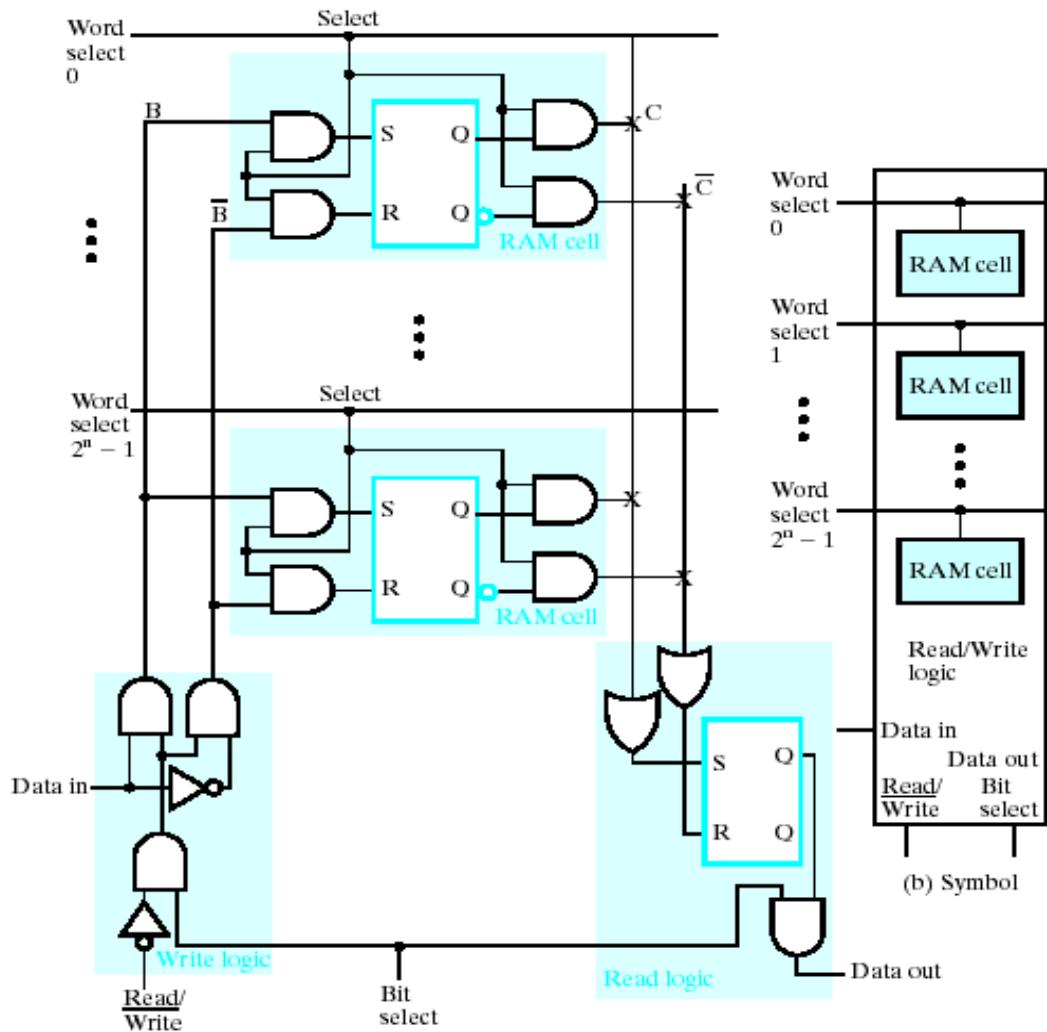


RAM Cell with SR Latch

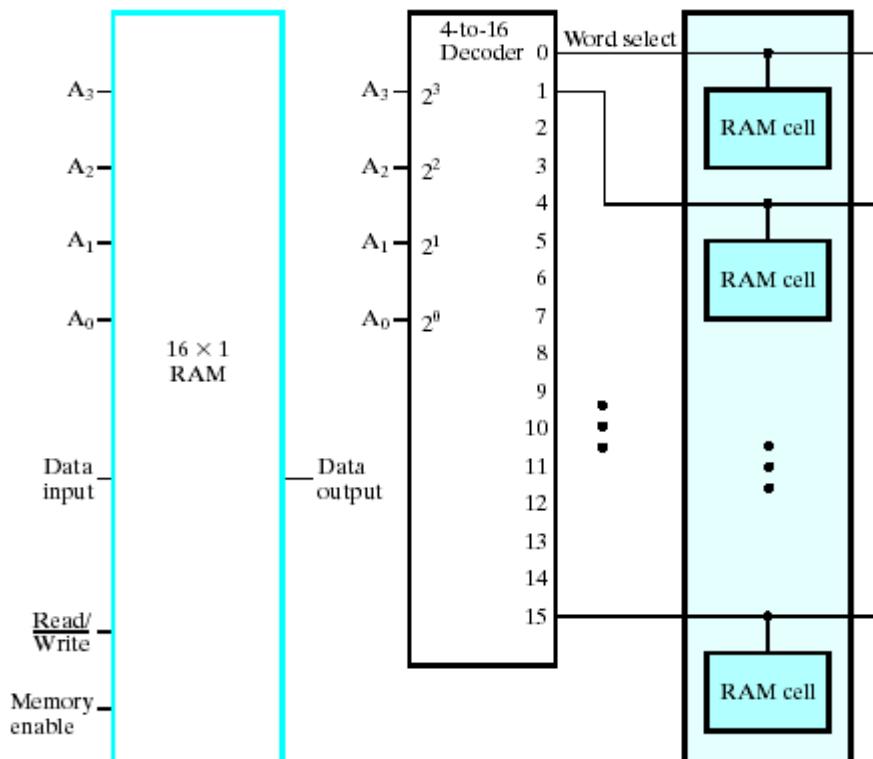


RAM Bit Slice Model

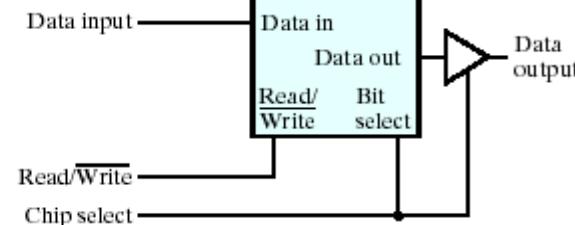
- RAM bit slice contains all of the circuitry associated with a single bit position of a set of RAM words



16-Word by 1-bit RAM Chip

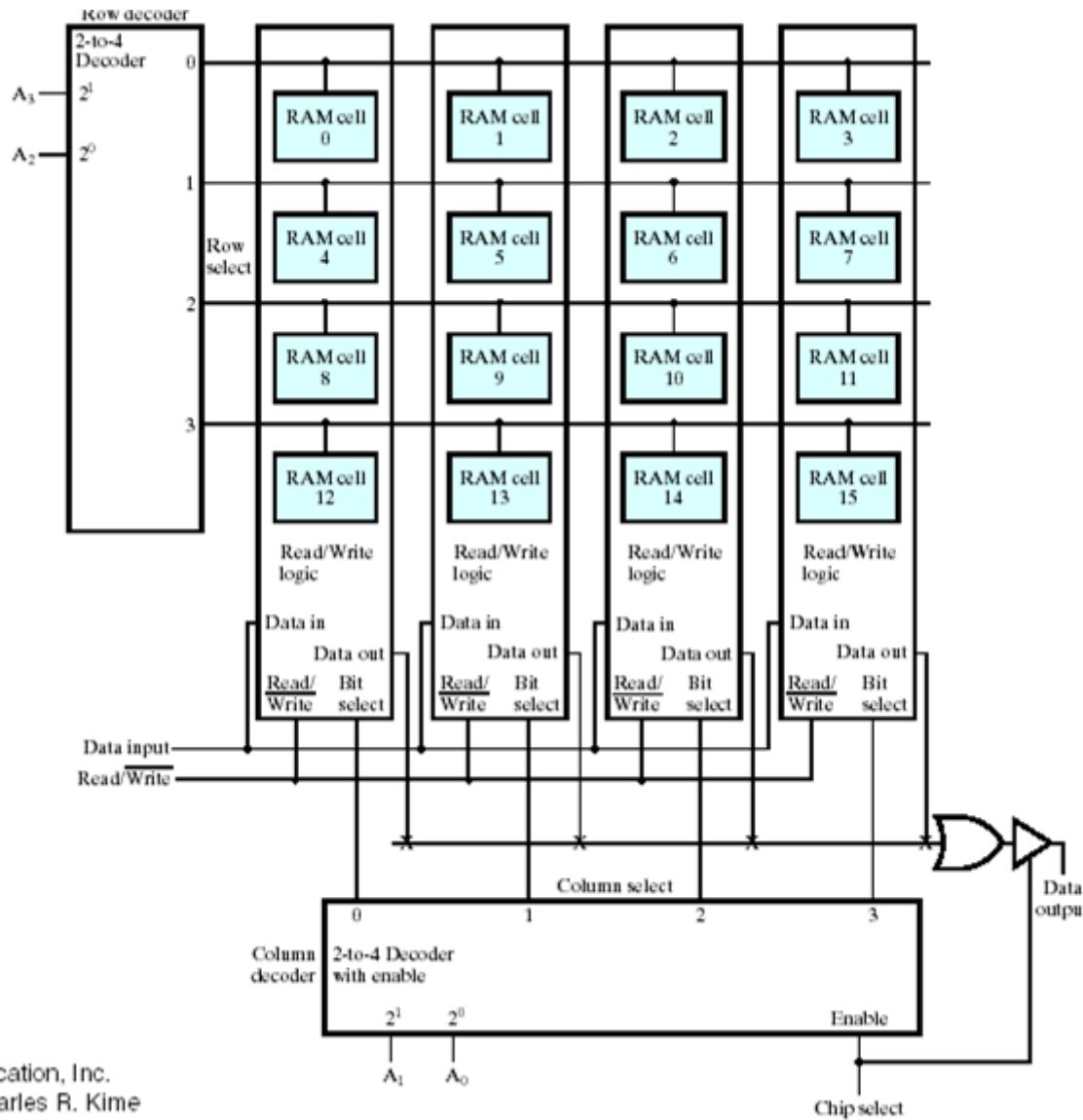


(a) Symbol

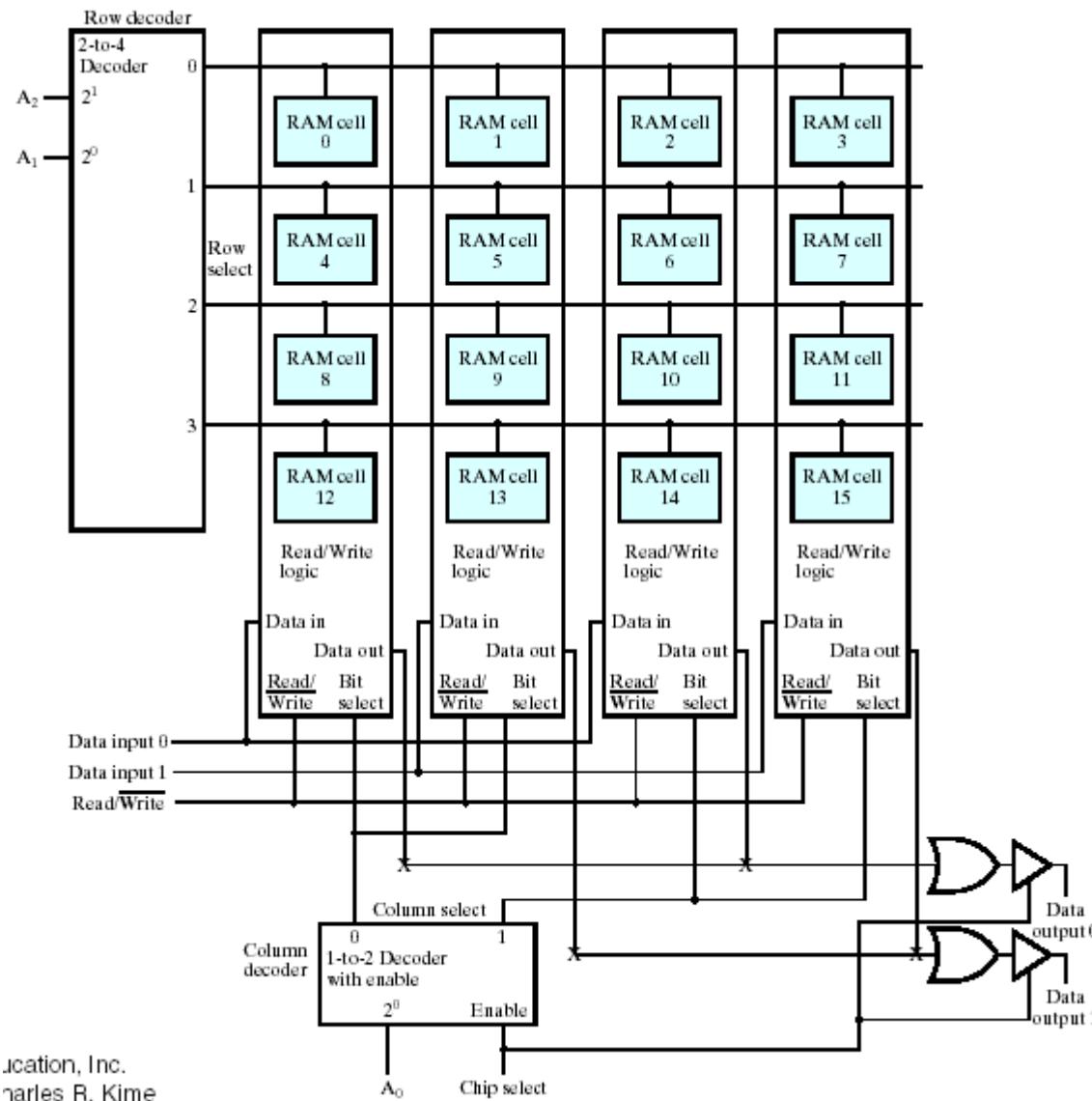


(b) Block diagram

16x1 RAM Using a 4x4 RAM Cell Array

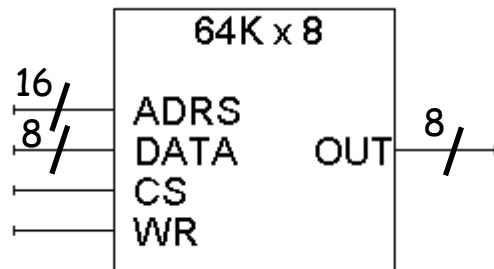


8x2 RAM Using a 4x4 RAM Cell Array



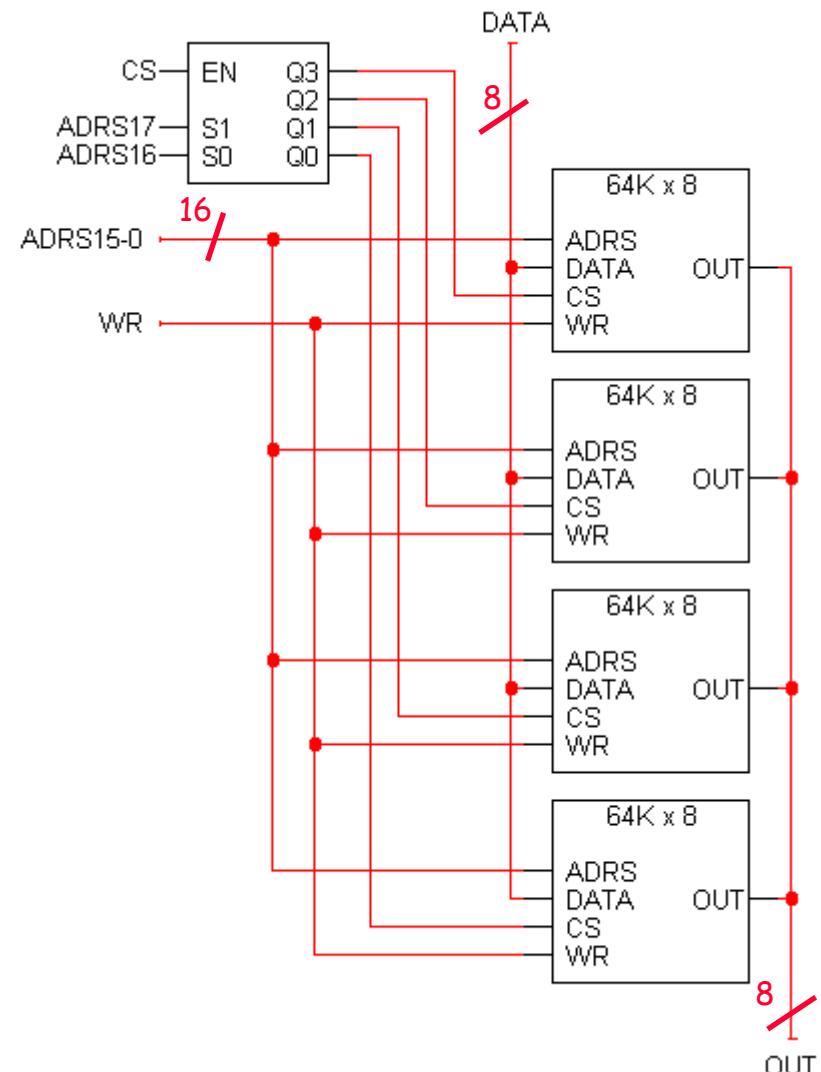
Bigger RAMs from smaller RAMs

- We can use small RAMs as building blocks for making larger memories, by following the same principles as in the previous examples.
- As an example, suppose we have some $64K \times 8$ RAMs to start with:
 - $64K = 2^6 \times 2^{10} = 2^{16}$, so there are 16 address lines.
 - There are 8 data lines.



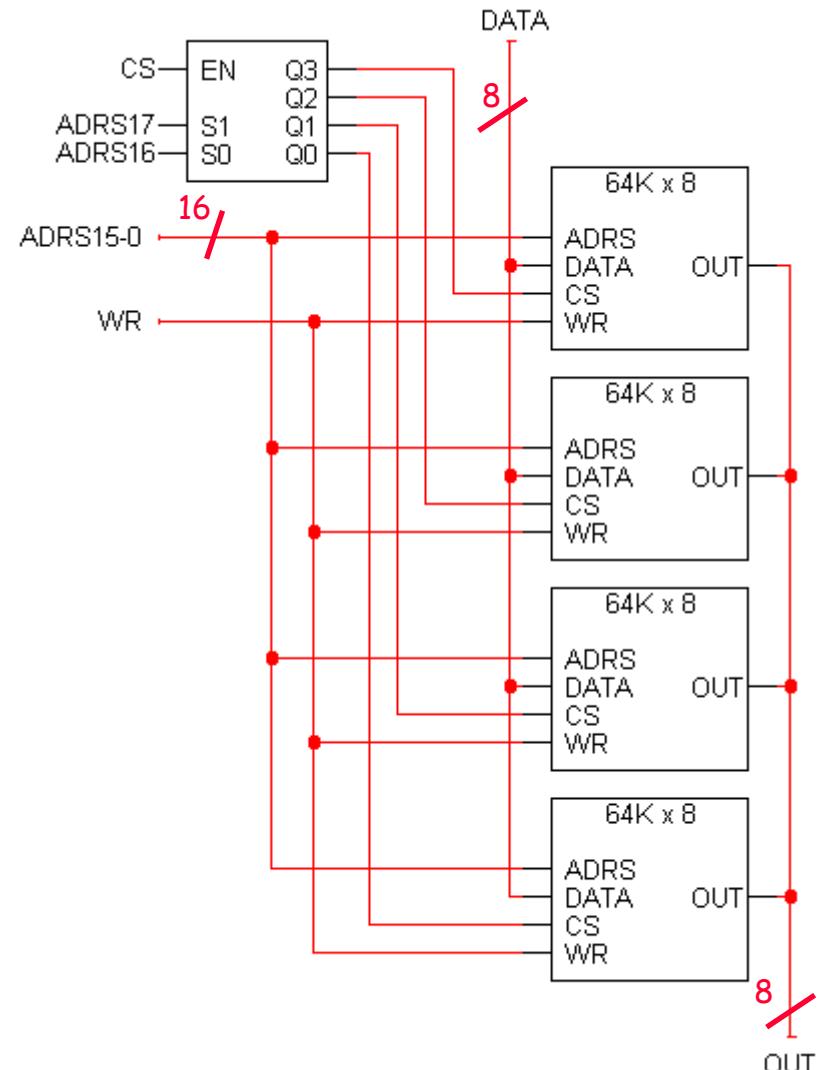
Making a larger memory

- We can put four $64K \times 8$ chips together to make a $256K \times 8$ memory.
- For $256K$ words, we need 18 address lines.
 - The two most significant address lines go to the decoder, which selects one of the four $64K \times 8$ RAM chips.
 - The other 16 address lines are shared by the $64K \times 8$ chips.
- The $64K \times 8$ chips also share WR and DATA inputs.
- This assumes the $64K \times 8$ chips have three-state outputs.

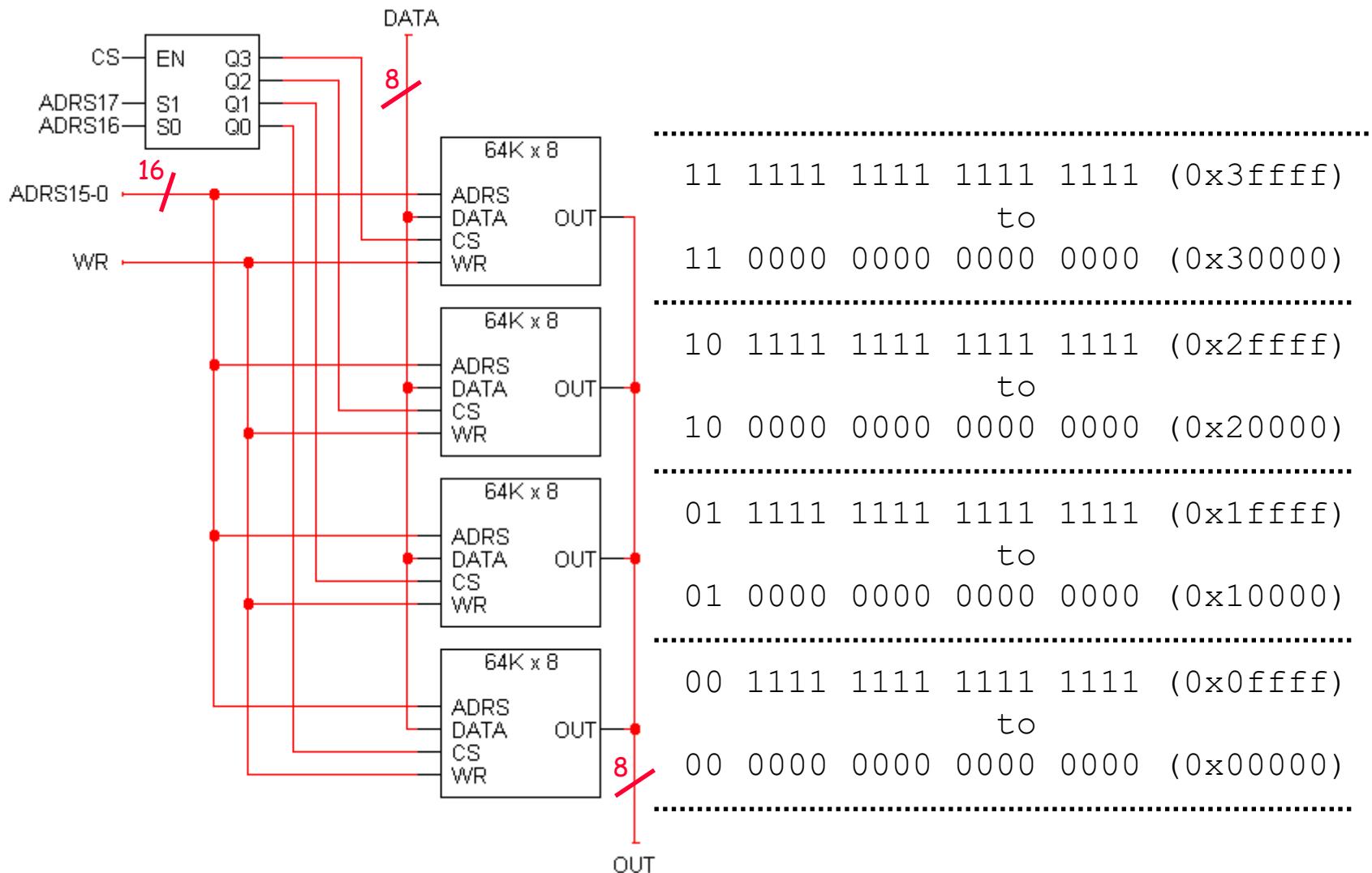


Analyzing the 256K × 8 RAM

- There are 256K words of memory, spread out among the four smaller 64K × 8 RAM chips.
- When the two most significant bits of the address are 00, the bottom RAM chip is selected. It holds data for the first 64K addresses.
- The next chip up is enabled when the address starts with 01. It holds data for the second 64K addresses.
- The third chip up holds data for the next 64K addresses.
- The final chip contains the data of the final 64K addresses.

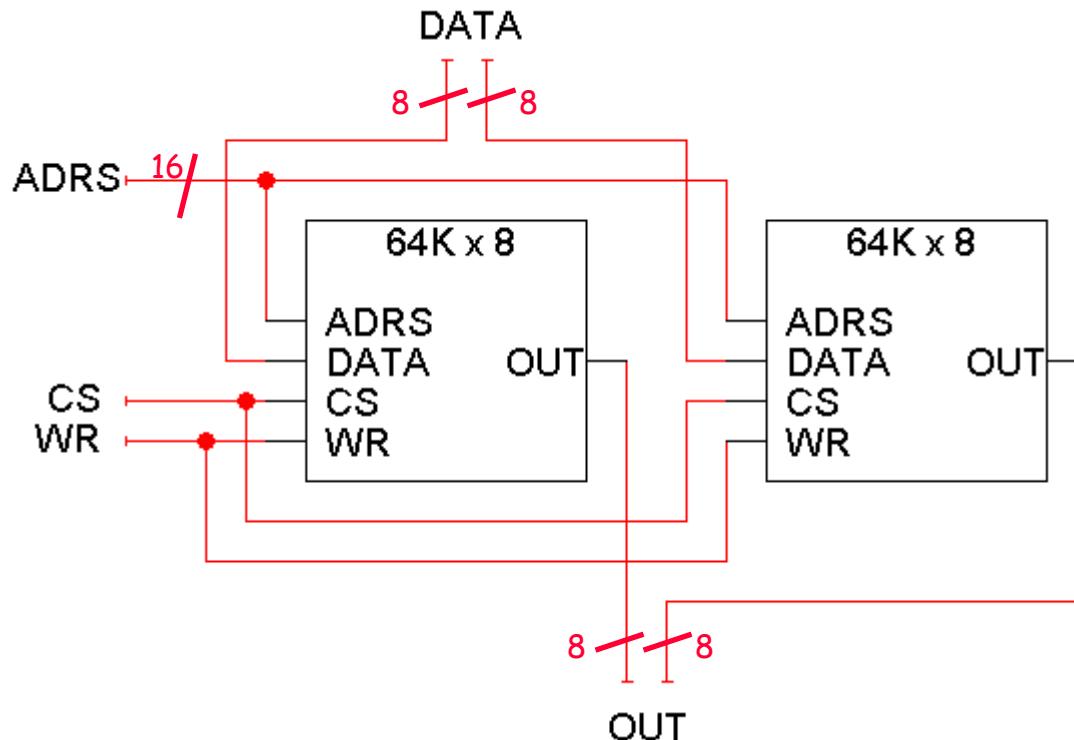


Address ranges



Making a wider memory

- You can also combine smaller chips to make wider memories, with the same number of addresses but more bits per word.
- Here is a 64K x 16 RAM, created from two 64K x 8 chips.
 - The left chip contains the most significant 8 bits of the data.
 - The right chip contains the lower 8 bits of the data.



Summary

- A RAM looks like a bunch of registers connected together, allowing users to select a particular address to read or write.
- Much of the hardware in memory chips supports this selection process:
 - Chip select inputs
 - Decoders
 - Tri-state buffers
- By providing a general interface, it's easy to connect RAMs together to make "longer" and "wider" memories.
- Next, we'll look at some other types of memories