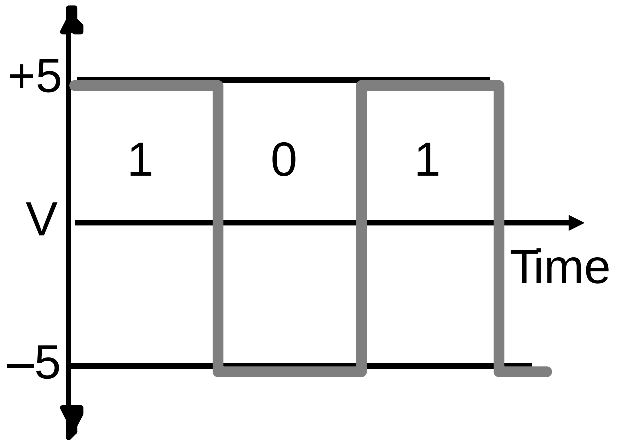
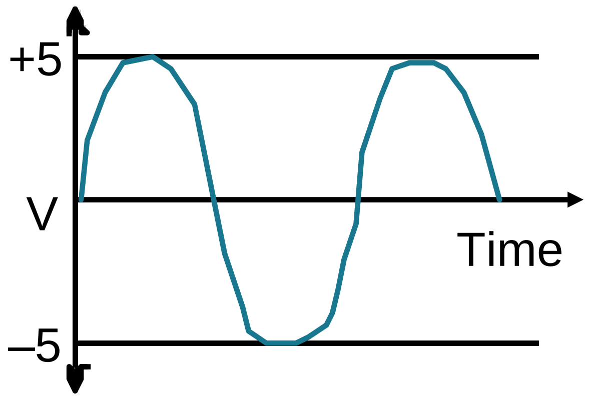
Digital Hardware Systems

Digital Systems

Digital vs. Analog Waveforms





Digital:
only assumes discrete values

Analog:
values vary over a broad range continuously

Digital Hardware Systems

- Digital Binary System
  - Two discrete values:
    - yes, on, 5 volts, current flowing, "1"
    - no, off, 0 volts, no current flowing, "0”
  - Advantage of binary systems:
    - rigorous mathematical foundation based on logic
    - it’s easy to implement

IF the garage door is open
AND the car is running
THEN the car can be backed out of the garage

both the door must be open and the car running before I can back out

the preconditions must be true to imply the conclusion

Binary Bit and Group Definitions

- Bit - a single binary digit
- Nibble - a group of four bits
- Byte - a group of eight bits
- Word - depends on processor; 8, 16, 32, or 64 bits
- LSB - Least Significant Bit (on the right)
- MSB - Most Significant Bit (on the left)

Binary Representation of Information

- Information divided into groups of symbols
  - 26 English letters
  - 10 decimal digits
  - 50 states in USA
- Digital systems manipulate information as 1’s & 0’s
- The mapping of symbols to binary value is known as a “code”
- The mapping must be unique

Minimum number of bits

- In binary, ‘r’ bits can represent $n = 2^r$ symbols
  - e.g. 3 bits can represent up to 8 symbols, 4 for 16, etc.
  - For N symbols to be represented, the minimum number of bits required is the lowest integer ‘r’ that satisifies the relationship:

$$2^r \geq N$$

e.g. if N = 26, minimum r is 5 since

$2^4 = 16$

$2^5 = 32$

Positional Number Systems

- Numeric value is represented by a series of digits
  - Number of digits used is fixed by radix
  - Digits multiplied by a power of the radix
  - Digit order determines radix powers
- Very large numbers can be represented
- Can also represent fractional values.

Positional Integer Number Values

Given a digit series of

$$A_{n-1} \cdots A_3 A_2 A_1 A_0 \bullet (\text{Radix point})$$

The full expression for the represented value is

$$A_{n-1} \times r^{n-1} + \cdots A_3 \times r^3 + A_2 \times r^2 + A_1 \times r^1 A_0 \times r^0$$

or

$$\sum_{i=0}^{i=n-1} A_i \times r^i$$

Positional Fractional Number Values

Given a digit series of

$$\text{(Radix point)} \bullet A_{-1} A_{-2} A_{-3} A_{-4} \cdots A_{-m}$$

The full expression for the represented value is

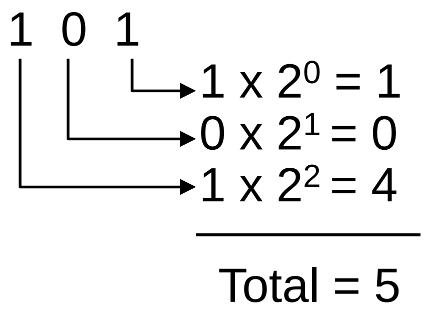
$$A_{-1} \times r^{-1} + A_{-2} \times r^{-2} + A_{-3} \times r^{-3} A_{-4} \times r^{-4} + \cdots A_{-m} \times r^{-m}$$

or

$$\sum_{i=-1}^{i=-m} A_i \times r^{i}$$

Binary Number System

- Just like decimal numbers except
  - The only valid digits are 0 and 1
  - The base is 2 instead of 10
- Binary to decimal conversion is just the explicit expression of the positional values,
- both integer and fraction
  - E.G.



Decimal to Binary Conversion

- Effectively the reverse of binary to decimal conversion
  - Integers:
    - Divide number by two; keep track of remainder
    - Repeat with dividend = last quotient until zero
    - First remainder is binary LSB, last is the MSB
  - Fractions:
    - Multiply fraction by two; keep track of integer part
    - Repeat with multiplier = last product fraction
    - First integer is MSB, last is the LSB
    - Conversion may not be exact; a repeated fraction

Decimal to Binary Conversion (cont.)

E.G. 13.2 to binary

Integer	Fraction
13 / 2 = 6 R 1 LSB	.2 x 2 = 0.4 MSB
6 / 2 = 3 R 0	.4 x 2 = 0.8
3 / 2 = 1 R 1	.8 x 2 = 1.6
1 / 2 = 0 R 1 MSB	.6 x 2 = 1.2
	.2 x 2 = 0.4 LSB repeating

Result is 1101.00110011……

If you're not sure of the results, convert back to decimal to check yourself.

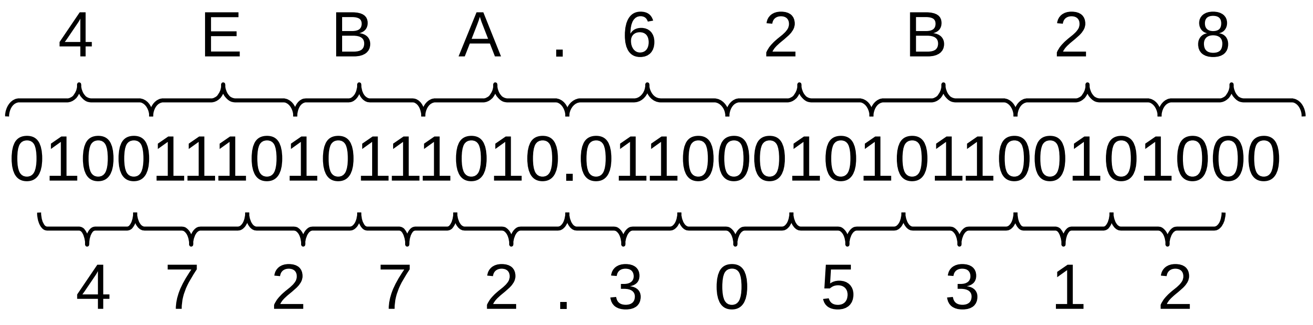
Octal and Hexadecimal Number Systems

- Both are positional systems with different radix and digits
  - Octal:
    - Radix = 8
    - Digits = 0,1,2,3,4,5,6,7
  - Hexadecimal:
    - Radix = 16
    - Digits = 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- Primary advantage of both is it's easy to convert to/from binary

Octal and Hexadecimal Conversions

- To/From decimal is same technique with a radix of 8 or 16 instead of 2
- To convert from binary:
  - Starting at radix point, go left/right and group bits into groups of 3 or 4 bits / group
  - Convert each bit group into equivalent octal or hex digit
- To convert to binary expand each octal / hex digit into equivalent 3 or 4 bit binary value.

Octal, Hex Conversion Example



Numeric Information Representation

- Numeric information has some special characteristics which influence the was it is represented
  - Number set is usually in positional notation
  - There is a defined range of numbers
  - There is a specified resolution for the set
- In general, numeric representations:
  - are in some form of positional binary notation
  - have no. of bits determined by range and res.

Numeric Representations (cont.)

- The number of values in the set of numbers is found from the following equation

$$N_{VALUES} = \frac{R_{MAX} - R_{MIN}}{RES} + 1$$

where R_{MAX} and R_{MIN} are the maximum and minimum range values and RES is the resolution

- The minimum number of bits needed must meet the relationship already presented

Numeric Representations (cont.)

- For example, the set of numbers from -5 to +10 with a resolution of 1 has 16 values

$$[+15 -(-5)] / 1 = 16$$

- Therefore the minimum number of bits is 4

$$2^4 = 16$$

Numeric Representations (cont.)

- For the set of numbers from 0 to 100 with a resolution of 10 we have 11 values

0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100

- For the set of numbers from 0 to 5 with a resolution of 0.1 we have 51 values

$$[(5 - 0) / 0.1] + 1 = 51$$

Numeric Representations (cont.)

- The actual representation could be any unique binary assignment but is usually of a positional form
  - binary *integer.fraction* with sufficient bits to meet the range and resolution criteria
  - binary *integer* form where the number of bits is as previously defined and the LSB value is the desired resolution

Numeric Representations (cont.)

- EG: Represent 0 to 5, resolution = 0.1
  - integer.fraction notation implies 3 bits for the integer (6 values) and 4 bits for the fraction (2^{-4} = 0.0625) for a total of 7 bits

    2.3 represented by 010.0101 (closest fraction)
  - integer * res notation requires 51 values or 6 bits; each value in set is represented by the equivalent binary integer = value / res

    2.3 represented by binary 010111 (2.3 / 0.1)

Numeric Representations (cont.)

- Negative ranges are handled by special assignments or negative number representations
- These are the most common numeric representations BUT they are certainly not the only ones!

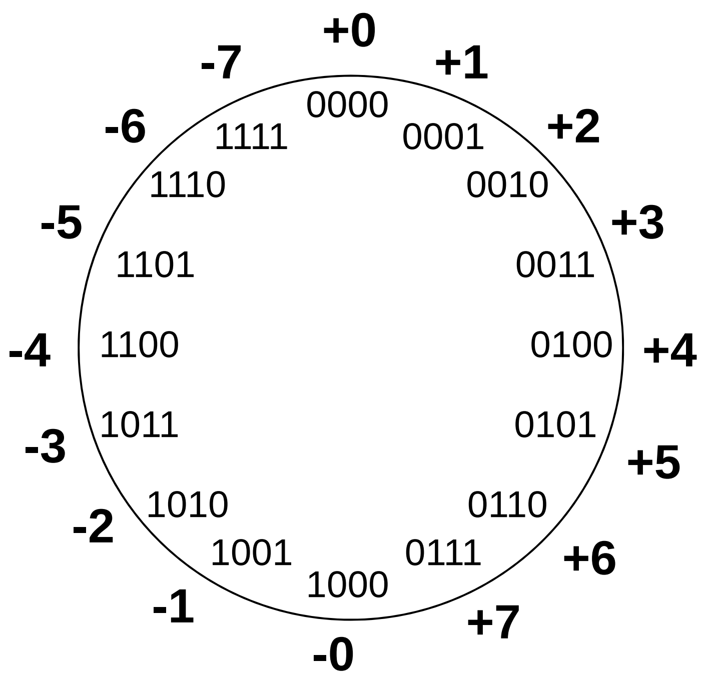
Representation of Signed Numbers

- Positive number representation same in most systems
  - Standard positional binary notation
  - MSB is the sign bit; 0 = plus, 1 = minus
- Major differences are in how negative numbers are represented
- Three major schemes:
  - sign and magnitude
  - ones complement
  - twos complement

Negative Number Representation

- Assumptions:
  - we'll assume a 4 bit machine word
  - 16 different values can be represented
  - roughly half are positive, half are negative
  - sign bit is the MSB; 0 = plus, 1 = minus

Sign-Magnitude Representation



High order bit is sign: 0 = positive (or zero), 1 = negative

Three low order bits is the magnitude: 0 (000) thru 7 (111)

Number range for n bits = ±2^{n-1} - 1

Two representations for 0

The major disadvantage is that we need separate circuits to both add and subtract

0 100 = +4
1 100 = - 4

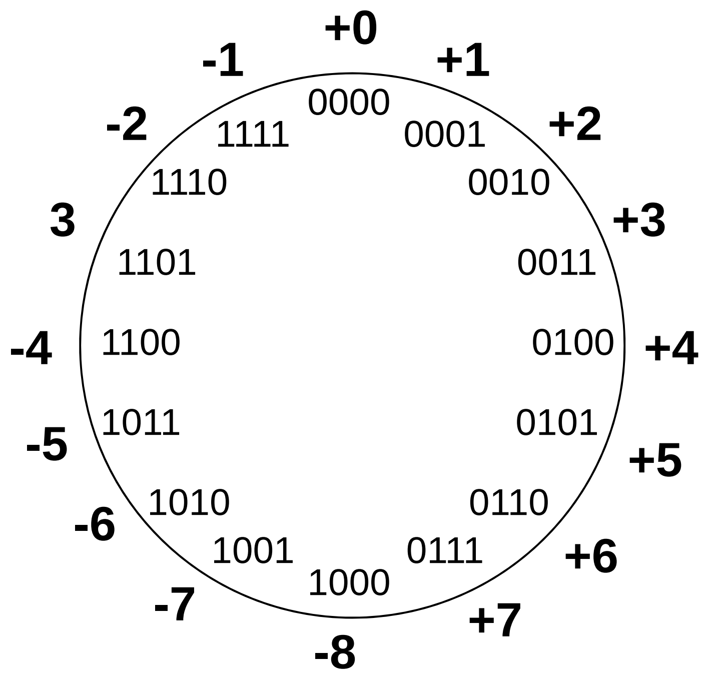
Number magnitudes need to be compared to get the right result

Representing -N

- What we really want is -N
  - Do A - B as A + (-B)
- We really are working in a closed, modulo number system; 0 to 2^{r-1} values
- Therefore for r bits, $2^r \equiv 0$
- If $-N \equiv 0 - N$ then $-N \equiv 2^r - N$

<u>This is the 2's complement representation for -N</u>

Twos Complement Representation



- Only one representation for 0
- One more negative number than positive number
- Generation of the 2’s complement as 2^r - N implies r + 1 bits available in system

Twos Complement Operations

Example: Twos complement of 7

$$\begin{array}{rl} 2^4 & = 10000 \\ \text{sub } 7 & = \underline{\ \ 0111\ } \\ & \ \ \ \ \ 1001 = \text{repr. of } -7 \end{array}$$

Example: Twos complement of -7

$$\begin{array}{rl} 2^4 & = 10000 \\ \text{sub } -7 & = \underline{\ \ 1001\ } \\ & \ \ \ \ \ 0111 = \text{repr. of } 7 \end{array}$$

Shortcut method:

Twos complement = bitwise complement + 1

$N^* = 2^r - N$

0111 -> 1000 + 1 -> 1001 (representation of -7)

1001 -> 0110 + 1 -> 0111 (representation of 7)

Ones Complement Representation

Ones Complement

N is positive number, then $\overline{N}$ is its negative 1's complement

$$\overline{N} = (2^n - 1) - N$$

Example: 1's complement of 7

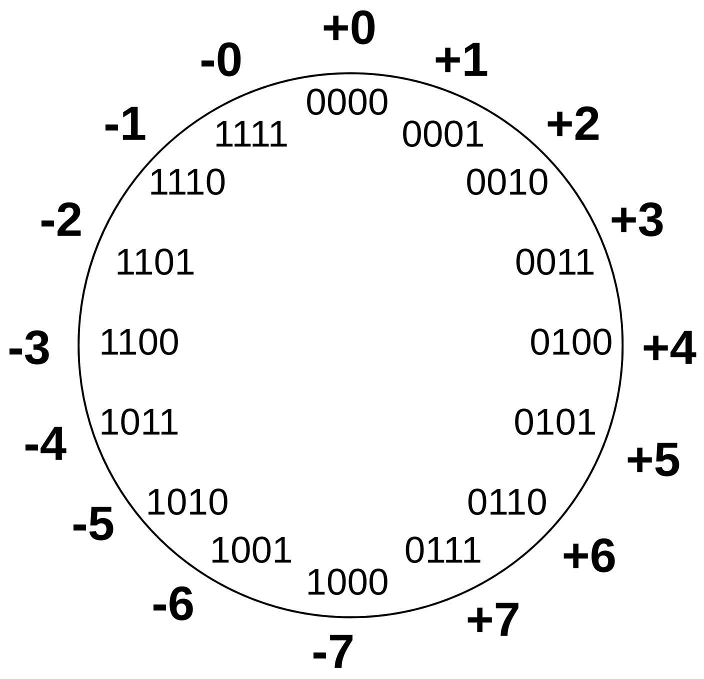
$$\begin{array}{rcr} 2^4 & = & 10000 \\ -1 & = & \underline{00001} \\ & & 1111 \\ -7 & = & \underline{0111} \\ & & 1000 \end{array} \quad = -7 \text{ in 1's comp.}$$

Shortcut method:

simply compute bit wise complement

0111 -> 1000

Ones Complement Representation



like 2's comp except shifted one position counter-clockwise

- Subtraction implemented by addition & 1's complement
- Still two representations of 0! This causes some problems
- Some complexities in addition

Addition and Subtraction of Numbers

Sign and Magnitude

result sign bit is the same as the operands' sign

```
    4    0100            -4    1100
  + 3    0011        + (-3)    1011
  ---    ----        ------    ----
    7    0111            -7    1111
```

when signs differ, operation is subtract, sign of result depends on sign of number with the larger magnitude

```
    4    0100            -4    1100
  - 3    1011           + 3    0011
  ---    ----           ---    ----
    1    0001            -1    1001
```

Addition and Subtraction of Numbers

Ones Complement Calculations

```
    4    0100                        -4      1011
  + 3    0011                     + (-3)     1100
  ----   ----                       ----    -----
    7    0111                        -7     10111
                                             |
                      End around carry       └──> 1
                                            -----
                                             1000

    4    0100                        -4    1011
  - 3    1100                       + 3    0011
  ----   ----                       ----   ----
    1   10000                        -1    1110
        |
End around carry └──> 1
        -----
         0001
```

Addition and Subtraction of Numbers

Ones Complement Calculations

Why does end-around carry work?

Its equivalent to subtracting 2^n and adding 1

$$M - N = M + \overline{N} = M + (2^n - 1 - N) = (M - N) + 2^n - 1 \qquad (M > N)$$

$$-M + (-N) = \overline{M} + \overline{N} = (2^n - M - 1) + (2^n - N - 1) \qquad M + N < 2^{n-1}$$

$$= 2^n + [2^n - 1 - (M + N)] - 1$$

after end around carry:

$$= 2^n - 1 - (M + N)$$

this is the correct form for representing -(M + N) in 1's comp!

Addition and Subtraction of Numbers

Twos Complement Calculations

$$\begin{array}{rr} 4 & 0100 \\ \underline{+\,3} & \underline{0011} \\ 7 & 0111 \end{array} \qquad \begin{array}{rr} -4 & 1100 \\ +\,\underline{(-3)} & \underline{1101} \\ -7 & 11001 \end{array}$$

If carry-in to sign = carry-out then ignore carry

if carry-in differs from carry-out then overflow

$$\begin{array}{rr} 4 & 0100 \\ \underline{-\,3} & \underline{1101} \\ 1 & 10001 \end{array} \qquad \begin{array}{rr} -4 & 1100 \\ \underline{+\,3} & \underline{0011} \\ -1 & 1111 \end{array}$$

Simpler addition scheme makes twos complement the most common choice for integer number systems within digital systems

Addition and Subtraction of Numbers

Twos Complement Calculations

Why can the carry-out be ignored?

-M + N when N > M:

$$M^* + N = (2^n - M) + N = 2^n + (N - M)$$

Ignoring carry-out is just like subtracting 2^n

-M + -N where N + M < or = 2^{n-1}

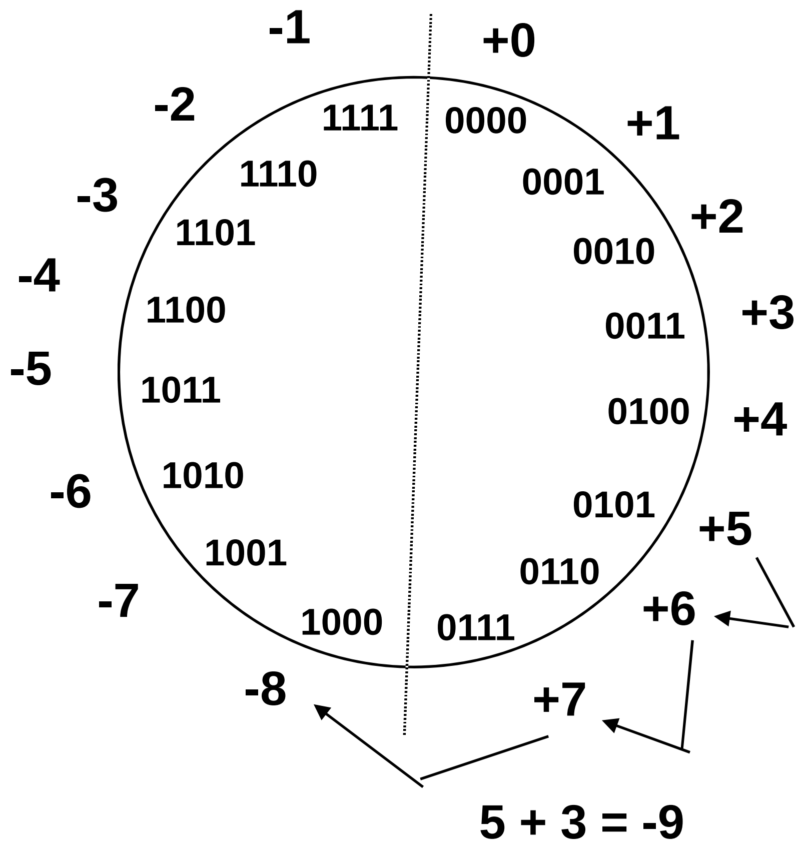
$$-M + (-N) = M^* + N^* = (2^n - M) + (2^n - N)$$

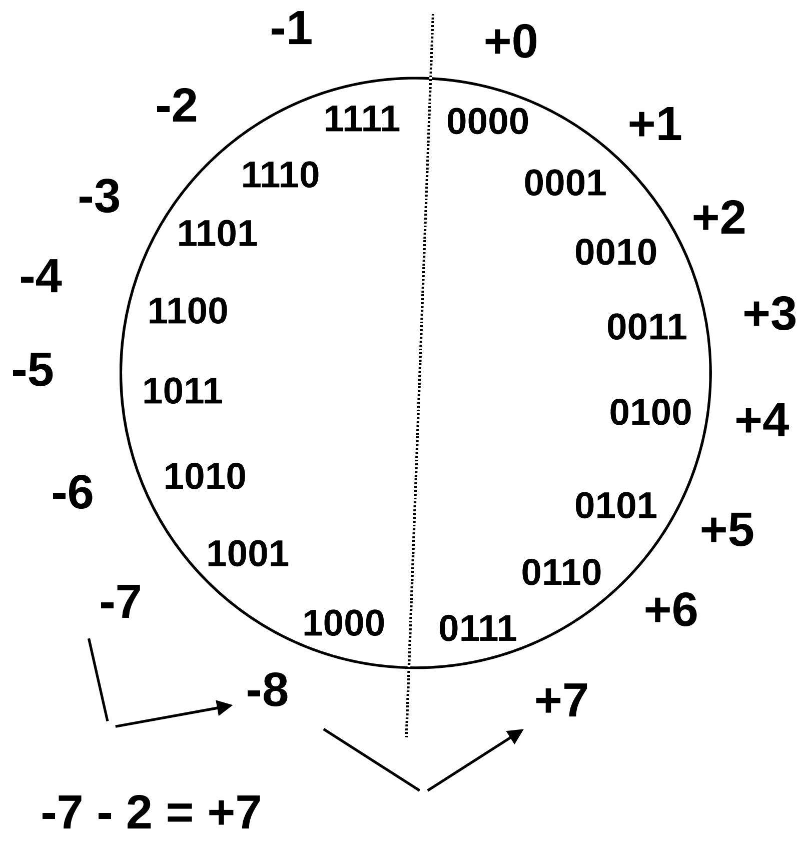
$$= 2^n - (M + N) + 2^n$$

After ignoring the carry, this is just the right twos compliment representation for -(M + N)!

Overflow Conditions

Add two positive numbers to get a negative number
or two negative numbers to get a positive number





Overflow Conditions

$$\begin{array}{r@{\qquad}ccccc}
 & 0 & 1 & 1 & 1 & \\
5 & & 0 & 1 & 0 & 1 \\
\underline{3} & & \underline{0} & \underline{0} & \underline{1} & \underline{1} \\
-8 & & 1 & 0 & 0 & 0
\end{array}$$

Overflow

$$\begin{array}{r@{\qquad}ccccc}
 & 1 & 0 & 0 & 0 & \\
-7 & & 1 & 0 & 0 & 1 \\
\underline{-2} & & \underline{1} & \underline{1} & \underline{0} & \underline{0} \\
7 & 1 & 0 & 1 & 1 & 1
\end{array}$$

Overflow

$$\begin{array}{r@{\qquad}ccccc}
 & 0 & 0 & 0 & 0 & \\
5 & & 0 & 1 & 0 & 1 \\
\underline{2} & & \underline{0} & \underline{0} & \underline{1} & \underline{0} \\
7 & & 0 & 1 & 1 & 1
\end{array}$$

No overflow

$$\begin{array}{r@{\qquad}ccccc}
 & 1 & 1 & 1 & 1 & \\
-3 & & 1 & 1 & 0 & 1 \\
\underline{-5} & & \underline{1} & \underline{0} & \underline{1} & \underline{1} \\
-8 & 1 & 1 & 0 & 0 & 0
\end{array}$$

No overflow

Overflow when carry in to sign does not equal carry out

Weighted and Unweighted Codes

- Most numeric number representations are in a class known as “Weighted Codes” where

$$\text{Value} = \sum_{i=0}^{r-1} b_i \bullet w_i$$

- Binary integers and fractions are special case where weights are powers of 2
- Unweighted codes are codes that cannot be assigned a weight value for each bit

Binary Coded Decimal

- Four bits are used to represent each decimal digit
  - In each 4-bit group, 6 values are not used
  - Many possible codes, natural BCD (equivalent binary digits) most common
  - BCD is not as efficient as binary
- BCD is easy to convert to/from decimal (it really is decimal with different symbols)
- BCD add/subtract circuits are complex

BCD Code Examples

Weighted codes

Unweighted code

Digit	8421	84-2-1	XS3
0	0000	0000	0011
1	0001	0111	0100
2	0010	0110	0101
3	0011	0101	0110
4	0100	0100	0111
5	0101	1011	1000
6	0110	1010	1001
7	0111	1001	1010
8	1000	1000	1011
9	1001	1111	1100

The 8421 or natural BCD code is the most common BCD code in use

BCD Addition

Case 1:

$$\begin{array}{r} 0001 \\ \underline{\quad 0101} \\ (0)\ 0110 \end{array} \qquad \begin{array}{r} 1 \\ \underline{\quad 5} \\ (0)\ 6 \end{array}$$

Case 2:

$$\begin{array}{r} 0110 \\ \underline{\quad 0101} \\ (0)\ 1011 \end{array} \qquad \begin{array}{r} 6 \\ \underline{\quad 5} \\ (1)\ 1 \end{array}$$

WRONG!

Case 3:

$$\begin{array}{r} 1000 \\ \underline{\quad 1001} \\ (1)\ 0001 \end{array} \qquad \begin{array}{r} 8 \\ \underline{\quad 9} \\ (1)\ 7 \end{array}$$

Note that for cases 2 and 3, adding a factor of 6 (0110) gives us the correct result.

BCD Addition (cont.)

- BCD addition is therefore performed as follows
  - 1) Add the two BCD digits together using normal binary addition
  - 2) Check if correction is needed
    - a) 4-bit sum is in range of 1010 to 1111
    - b) carry out of MSB = 1
  - 3) If correction is required, add 0110 to 4-bit sum to get the correct result; BCD carry out = 1

BCD Negative Number Representation

- Similar to binary negative number representation except r = 10.
  - BCD sign-magnitude
    - MSD (sign digit options)
      - MSD = 0 (positive); not equal to 0 = negative
      - MSD range of 0-4 positive; 5-9 negative
  - BCD 10's complement
    - -N $\equiv 10^r$ - N; 9's complement + 1
  - BCD 9;s complement
    - invert each BCD digit (0$\rightarrow$9, 1 $\rightarrow$ 8, 2 $\rightarrow$ 7,3 $\rightarrow$ 6, …7 $\rightarrow$ 2, 8 $\rightarrow$ 1, 9 $\rightarrow$ 0)

Negative BCD Numbers

- 84-2-1 and XS3 codes allow for easy digit inversion.
- XS3 code is also easy to implement
  - Addition is like binary
  - Correction factor is -3 or +3

Gray Codes

- Grey codes are *minimum change* codes
  - From one numeric representation to the next, only one bit changes
  - Primary use is in numeric input encoding apps. where we expect non-random input values changes (I.e. value n to either n-1 or n+1)
    - Milling machine table position
    - Rotary shaft position

Gray Codes (cont.)

Binary	Grey
0000	0000
0001	0001
0010	0011
0011	0010
0100	0110
0101	0111
0110	0101
0111	0100
1000	1100
1001	1101
1010	1111
1011	1110
1100	1010
1101	1011
1110	1001
1111	1000

Alphanumeric Representation

- Binary codes used to represent alphabetic and numeric characters
- Two most common are:
  - ASCII, 7 bit code, 128 symbols
  - EBCDIC, 8 bit code, 256 symbols
- Problems can arise when comparing symbol values (collation)
  - Comparing ‘A’ to ‘a’ in ASCII system yields different results in an EBCDIC system.

Parity Bit

- ASCII code may have an extra bit appended to detect data transmission errors
  - P = 0 if the number of 1s in the character is even, else P = 1 (even parity)
  - P = 0 if the number of 1s in the character is odd, else P = 1 (odd parity)
- If any single bit changes, parity will be wrong at receive end

	Even parity	Odd parity
ASCII A = 1000001	01000001	11000001
ASCII T = 1010100	11010100	01010100

Other Information Representation

- ALL information must be encoded before we can design circuits to process it
- You can assign any code to any information
  - E.G. 00 - north, 01 - east, 11 - south, 10 - west
- If the information goes somewhere else, the user has to have access to your definition
- Standards are best if available
  - Already published and easily available
  - Allows your system to work with many others

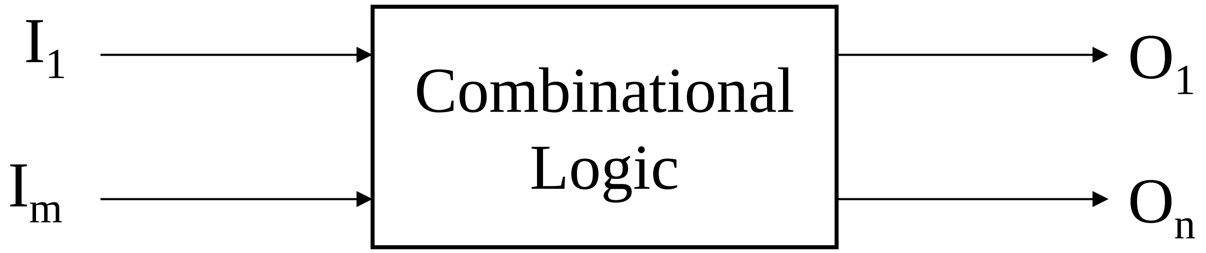
Combinational Logic Circuits

Overview

- Binary logic operations and gates
- Switching algebra
- Algebraic Minimization
- Standard forms
- Karnaugh Map Minimization
- Other logic operators
- IC families and characteristics

Combinational Logic

- One or more digital signal inputs
- One or more digital signal outputs
- Outputs are only functions of current input values (ideal) plus logic propagation delays



$$O_1(t+\Delta t) = F_1(I_1(t),\ldots I_m(t))$$

$$\vdots$$

$$O_n(t+\Delta t) = F_n(I_1(t),\ldots I_m(t))$$

Combinational Logic (cont.)

- Combinational logic has no memory!
  - Outputs are only function of current input combination
  - Nothing is known about past events
  - Repeating a sequence of inputs always gives the same output sequence
- Sequential logic (covered later) does have memory
  - Repeating a sequence of inputs can result in an entirely different output sequence

Switching Algebra

- Based on Boolean Algebra
  - Developed by George Boole in 1854
  - Formal way to describe logic statements and determine truth of statements
- Only has two-values domain (0 and 1)
- Huntington's Postulates define underlying assumptions

Huntington's Postulates

- Closure

  If X and Y are in set (0,1) then operations X+Y and X ·Y are also in set (0,1)

- Identity

  $X + 0 = X \qquad\qquad X \cdot 1 = X$

- Commutative

  $X + Y = Y + X \qquad\qquad X \cdot Y = Y \cdot X$

Huntington's Postulates (cont.)

- Distributive

$$X \cdot (Y + Z) = (X \cdot Y) + (X \cdot Z)$$

$$X + (Y \cdot Z) = (X + Y) \cdot (X + Z)$$

- Complement

$$X + \overline{X} = 1$$

$$X \cdot \overline{X} = 0$$

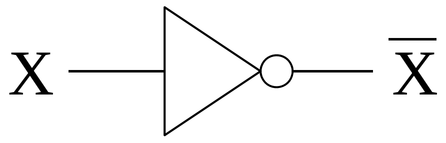
Note that for each property, one form is the dual of the other;

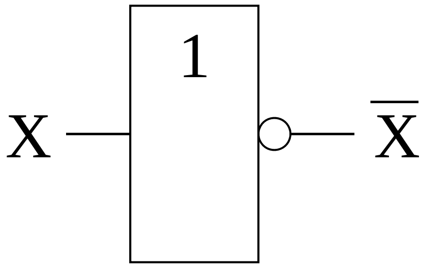
(0s to 1s, 1s to 0s, ·s to +s, +s to ·s)

Switching Algebra Operations - Not

- Unary complement or inversion operation
- Usually shown as overbar ($\overline{X}$), other forms are ~X, X'

X	$\overline{X}$
0	1
1	0

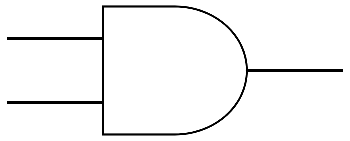


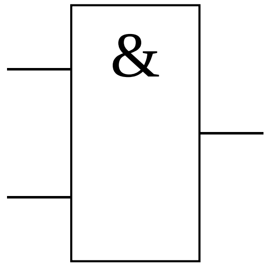


Switching Algebra Operations - AND

- Also known as the conjunction operation; output is true (1) only if all inputs are true
- Algebraic operators are '·', '&', '∧'

X	Y	X·Y
0	0	0
0	1	0
1	0	0
1	1	1

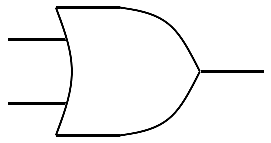


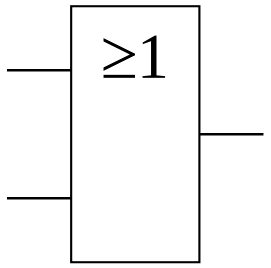


Switching Algebra Operations - OR

- Also known as the disjunction operation; output is true (1) if any input is true
- Algebraic operators are '+', '|', '∨'

X	Y	X+Y
0	0	0
0	1	1
1	0	1
1	1	1





Logic Expressions

- Terms and Definitions
  - Logic Expression - a mathematical formula consisting of logical operators and variables
  - Logic Operator - a function that gives a well defined output according to switching algebra
  - Logic Variable - a symbol representing the two possible switching algebra values of 0 and 1
  - Logic Literal - the values 0 and 1 or a logic variable or it’s complement

Logic Expressions - Precedence

- Like standard algebra, switching algebra operators have a precedence of evaluation
  - NOT operations have the highest precedence
  - AND operations are next
  - OR operations are lowest
- Parentheses explicitly define the order of operator evaluation
  - If in doubt, USE PARENTHESES!

Logic Expression Minimization

- Goal is to find an equivalent of an original logic expression that:
  - a) has fewer variables per term
  - b) has fewer terms
  - c) needs less logic to implement
- There are three main manual methods
  - Algebraic minimization
  - Karnaugh Map minimization
  - Quine-McCluskey (tabular) minimization

Algebraic Minimization

- Process is to apply the switching algebra postulates, laws, and theorems to transform the original expression
  - Hard to recognize when a particular law can be applied
  - Difficult to know if resulting expression is truly minimal
  - Very easy to make a mistake
    - Incorrect complementation
    - Dropped variables

Switching Algebra Laws and Theorems

Involution:

$$X = \overline{\left(\overline{X}\right)}$$

Switching Algebra Laws and Theorems

Identity:

$$X + 1 = 1 \qquad X \cdot 0 = 0$$

$$X + 0 = X \qquad X \cdot 1 = X$$

Switching Algebra Laws and Theorems

Idempotence:

$$X + X = X \qquad\qquad X \cdot X = X$$

Switching Algebra Laws and Theorems

Associativity:

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

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

Switching Algebra Laws and Theorems

Adjacency:

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

$$(X + Y) \cdot \left(X + \overline{Y}\right) = X$$

Switching Algebra Laws and Theorems

Absorption:

$$X + (X \cdot Y) = X$$

$$X \cdot (X + Y) = X$$

Switching Algebra Laws and Theorems

Simplification:

$$X + \left(\overline{X} \cdot Y\right) = X + Y$$

$$X \cdot \left(\overline{X} + Y\right) = X \cdot Y$$

Switching Algebra Laws and Theorems

Consensus:

$$X \cdot Y + \overline{X} \cdot Z + Y \cdot Z = X \cdot Y + \overline{X} \cdot Z$$

$$\left(X + Y\right) \cdot \left(\overline{X} + Z\right) \cdot \left(Y + Z\right) = \left(X + Y\right) \cdot \left(\overline{X} + Z\right)$$

Switching Algebra Laws and Theorems

DeMorgan's Theorem:

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

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

General form:

$$\overline{F(\cdot, +, X_1, \ldots X_n)} = G(+, \cdot, \overline{X_1}, \ldots \overline{X_n})$$

DeMorgan's Theorem

Very useful for complementing function expressions:

e.g.

$$F = X + Y \cdot Z; \qquad \overline{F} = \overline{X + Y \cdot Z}$$

$$\overline{F} = \overline{X} \cdot \overline{Y \cdot Z} \qquad F = \overline{X} \cdot \left(\overline{Y} + \overline{Z}\right)$$

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

Minimization via Adjacency

- Adjacency is easy to use; very powerful
  - Look for two terms that are identical except for one variable

$$\text{e.g. } A \cdot B \cdot C \cdot \overline{D} \;+\; A \cdot B \cdot C \cdot D$$

  - Application removes one term and one variable from the remaining term

$$A \cdot B \cdot C \cdot \overline{D} \;+\; A \cdot B \cdot C \cdot D \;=\; A \cdot B \cdot C$$

$$(A \cdot B \cdot C) \cdot \overline{D} \;+\; (A \cdot B \cdot C) \cdot D \;=\; A \cdot B \cdot C$$

$$(A \cdot B \cdot C) \cdot \left(\overline{D} + D\right) \;=\; (A \cdot B \cdot C) \cdot 1 \;=\; A \cdot B \cdot C$$

Example of Adjacency Minimization

Adjacencies

$$x_3 = \overline{b_3}b_2\overline{b_1}b_0 + \overline{b_3}b_2b_1\overline{b_0} + \overline{b_3}b_2b_1b_0 + \overbrace{b_3\overline{b_2}\,\overline{b_1}\,\overline{b_0} + b_3\overline{b_2}\,\overline{b_1}b_0}$$

Duplicate 3rd. term and rearrange

$$x_3 = \overline{b_3}b_2\overline{b_1}b_0 + \overline{b_3}b_2b_1b_0 + \overline{b_3}b_2b_1\overline{b_0} + \overline{b_3}b_2b_1b_0 + b_3\overline{b_2}\,\overline{b_1}\,\overline{b_0} + b_3\overline{b_2}\,\overline{b_1}b_0$$

Apply adjacency on term pairs

$$x_3 = \overline{b_3}b_2b_0 + \overline{b_3}b_2b_1 + b_3\overline{b_2}\,\overline{b_1}$$

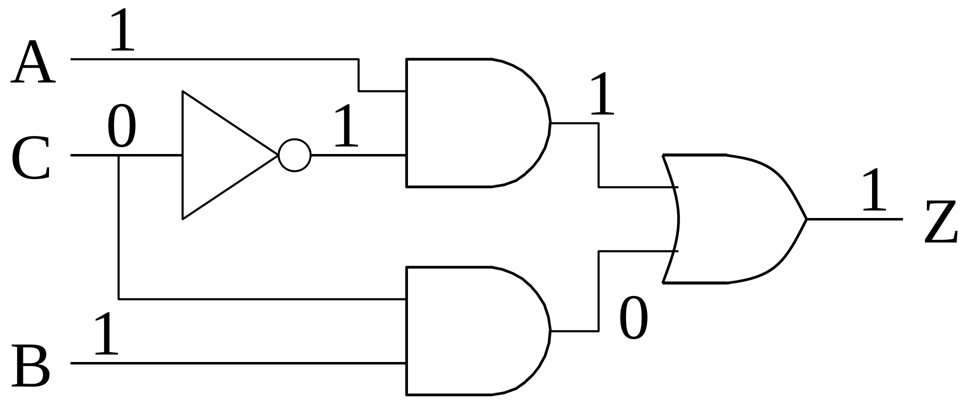
Combinational Circuit Analysis

- Combinational circuit analysis starts with a schematic and answers the following questions:
  - What is the truth table(s) for the circuit output function(s)
  - What is the logic expression(s) for the circuit output function(s)

Literal Analysis

- Literal analysis is process of manually assigning a set of values to the inputs, tracing the results, and recording the output values
  - For ‘n’ inputs there are 2^n possible input combinations
  - From input values, gate outputs are evaluated to form next set of gate inputs
  - Evaluation continues until gate outputs are circuit outputs
- Literal analysis only gives us the truth table

Literal Analysis - Example



A	B	C	Z
0	0	0	x
0	0	1	x
0	1	0	x
0	1	1	x
1	0	0	x
1	0	1	x
1	1	0	1
1	1	1	x

Assign input values

Determine gate outputs and propagate

Repeat until we reach output

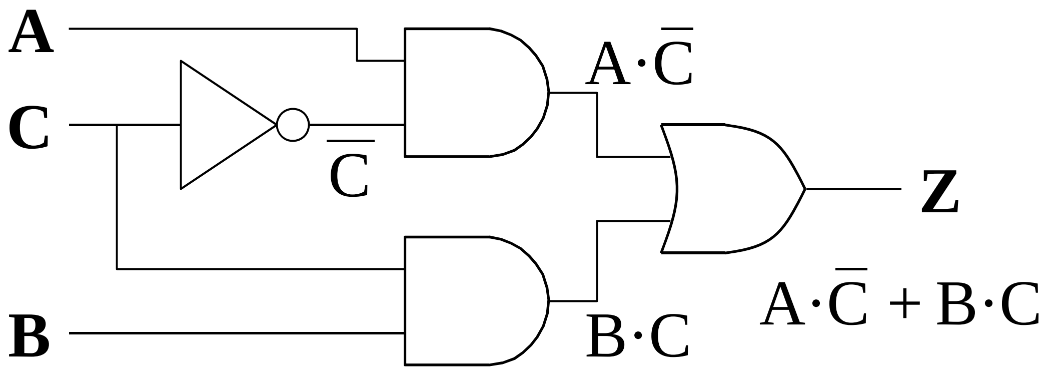
Symbolic Analysis

- Like literal analysis we start with the circuit diagram
  - Instead of assigning values, we determine gate output expressions instead
  - Intermediate expressions are combined in following gates to form complex expressions
  - We repeat until we have the output function and expression
- Symbolic analysis gives both the truth table and logic expression

Symbolic Analysis (cont.)

- Note that we are constructing the truth table as we go
    - truth table has a column for each intermediate gate output
    - intermediate outputs are combined in the truth table to generate the complex columns
- Symbolic analysis is more work but gives us complete information

Symbolic Analysis - Example



Generate intermediate expression

Create associated TT column

Repeat till output reached

A	B	C	$\overline{C}$	$A \cdot \overline{C}$	$B \cdot C$	$Z = A \cdot \overline{C} + B \cdot C$
0	0	0	1	0	0	0
0	0	1	0	0	0	0
0	1	0	1	0	0	0
0	1	1	0	0	1	1
1	0	0	1	1	0	1
1	0	1	0	0	0	0
1	1	0	1	1	0	1
1	1	1	0	0	1	1

Standard Expression Forms

- Two standard (canonical) expression forms
  - Canonical sum form
    - AKA disjunctive normal form or sum-of-products
    - OR of AND terms
  - Canonical product form
    - AKA conjunctive normal form or product-of-sums
    - AND or OR terms
- In both forms, each first-level operator corresponds to one row of truth table
- 2nd-level operator combines 1st-level results

Standard Forms (cont.)

Standard Sum Form

Sum of Products (OR of AND terms)

$$F[A,B,C] = \left(\overline{A}\cdot\overline{B}\cdot\overline{C}\right) + \left(\overline{A}\cdot B\cdot C\right) + \left(A\cdot B\cdot\overline{C}\right) + \left(A\cdot B\cdot C\right)$$

Minterms

Standard Product Form

Product of Sums (AND of OR terms)

$$F[A,B,C] = \left(A+B+\overline{C}\right)\cdot\left(A+\overline{B}+C\right)\cdot\left(\overline{A}+B+C\right)\cdot\left(\overline{A}+B+\overline{C}\right)$$

Maxterms

Standard Sum Form

- Each product (AND) term is a Minterm
  - ANDed product of literals in which each variable appears exactly once, in true or complemented form (but not both!)
  - Each minterm has exactly one ‘1’ in the truth table
  - When minterms are ORed together each minterm contributes a ‘1’ to the final function

NOTE: NOT ALL PRODUCT TERMS ARE MINTERMS!

Minterms and Standard Sum Form

A	B	C	Minterms	m_0	m_3	m_6	m_7	F
0	0	0	$m_0 = \overline{A}\cdot\overline{B}\cdot\overline{C}$	1	0	0	0	1
0	0	1	$m_1 = \overline{A}\cdot\overline{B}\cdot C$	0	0	0	0	0
0	1	0	$m_2 = \overline{A}\cdot B\cdot\overline{C}$	0	0	0	0	0
0	1	1	$m_3 = \overline{A}\cdot B\cdot C$	0	1	0	0	1
1	0	0	$m_4 = A\cdot\overline{B}\cdot\overline{C}$	0	0	0	0	0
1	0	1	$m_5 = A\cdot\overline{B}\cdot C$	0	0	0	0	0
1	1	0	$m_6 = A\cdot B\cdot\overline{C}$	0	0	1	0	1
1	1	1	$m_7 = A\cdot B\cdot C$	0	0	0	1	1

$$F = \overline{A}\cdot\overline{B}\cdot\overline{C} + \overline{A}\cdot B\cdot C + A\cdot B\cdot\overline{C} + A\cdot B\cdot C$$

$$F(A,B,C) = m_0 + m_3 + m_6 + m_7$$

$$F(A,B,C) = \sum m(0,3,6,7)$$

Standard Product Form

- Each OR (sum) term is a Maxterm
    - ORed product of literals in which each variable appears exactly once, in true or complemented form (but not both!)
    - Each maxterm has exactly one '0' in the truth table
    - When maxterms are ANDed together each maxterm contributes a '0' to the final function

NOTE: NOT ALL SUM TERMS ARE MAXTERMS!

Maxterms and Standard Product Form

A	B	C	Maxterms	M_1	M_2	M_4	M_5	F
0	0	0	$M_0 = A+B+C$	1	1	1	1	1
0	0	1	$M_1 = A+B+\overline{C}$	0	1	1	1	0
0	1	0	$M_2 = A+\overline{B}+C$	1	0	1	1	0
0	1	1	$M_3 = A+\overline{B}+\overline{C}$	1	1	1	1	1
1	0	0	$M_4 = \overline{A}+B+C$	1	1	0	1	0
1	0	1	$M_5 = \overline{A}+B+\overline{C}$	1	1	1	0	0
1	1	0	$M_6 = \overline{A}+\overline{B}+C$	1	1	1	1	1
1	1	1	$M_7 = \overline{A}+\overline{B}+\overline{C}$	1	1	1	1	1

$$F = \left(A+B+\overline{C}\right)\cdot\left(A+\overline{B}+C\right)\cdot\left(\overline{A}+B+C\right)\cdot\left(\overline{A}+B+\overline{C}\right)$$

$$F(A,B,C) = M_1 \cdot M_2 \cdot M_4 \cdot M_5$$

$$F(A,B,C) = \prod M(1,2,4,5)$$

BCD to XS3 Example

b_3	b_2	b_1	b_0	x_3	x_2	x_1	x_0
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0
1	0	1	0	-	-	-	-
1	0	1	1	-	-	-	-
1	1	0	0	-	-	-	-
1	1	0	1	-	-	-	-
1	1	1	0	-	-	-	-
1	1	1	1	-	-	-	-

Note: Don’t cares can work to our advantage during minimization; we can assign either 0 or 1 as needed. Assume 0’s for now.

BCD to XS3 Example (cont.)

- Generate the Standard Sum of Products logical expressions for the outputs

$$x_3 = \overline{b_3}b_2\overline{b_1}b_0 + \overline{b_3}b_2b_1\overline{b_0} + \overline{b_3}b_2b_1b_0 + b_3\overline{b_2}\,\overline{b_1}\,\overline{b_0} + b_3\overline{b_2}\,\overline{b_1}b_0$$
$$x_2 = \overline{b_3}\,\overline{b_2}\,\overline{b_1}b_0 + \overline{b_3}\,\overline{b_2}b_1\overline{b_0} + \overline{b_3}\,\overline{b_2}b_1b_0 + \overline{b_3}b_2\overline{b_1}\,\overline{b_0} + b_3\overline{b_2}\,\overline{b_1}b_0$$
$$x_1 = \overline{b_3}\,\overline{b_2}\,\overline{b_1}\,\overline{b_0} + \overline{b_3}\,\overline{b_2}b_1b_0 + \overline{b_3}b_2\overline{b_1}\,\overline{b_0} + \overline{b_3}b_2b_1b_0 + b_3\overline{b_2}\,\overline{b_1}\,\overline{b_0}$$
$$x_0 = \overline{b_3}\,\overline{b_2}\,\overline{b_1}\,\overline{b_0} + \overline{b_3}\,\overline{b_2}b_1\overline{b_0} + \overline{b_3}b_2\overline{b_1}\,\overline{b_0} + \overline{b_3}b_2b_1\overline{b_0} + b_3\overline{b_2}\,\overline{b_1}\,\overline{b_0}$$

Karnaugh Map Minimization

- Karnaugh Map (or K-map) minimization is a visual minimization technique
  - Is an application of adjacency
  - Procedure guarantees a minimal expression
  - Easy to use; fast
  - Problems include:
    - Applicable to limited number of variables (4 ~ 8)
    - Errors in translation from TT to K-map
    - Not grouping cells correctly
    - Errors in reading final expression

K-map Minimization (cont.)

- Basic K-map is a 2-D rectangular array of cells
  - Each K-map represents one bit column of output
  - Each cell contains one bit of output function
- Arrangement of cells in array facilitates recognition of adjacent terms
  - Adjacent terms differ in one variable value; equivalent to difference of one bit of input row values
    - e.g. m6 (110) and m7 (111)

Truth Table Rows and Adjacency

Standard TT ordering doesn't show adjacency

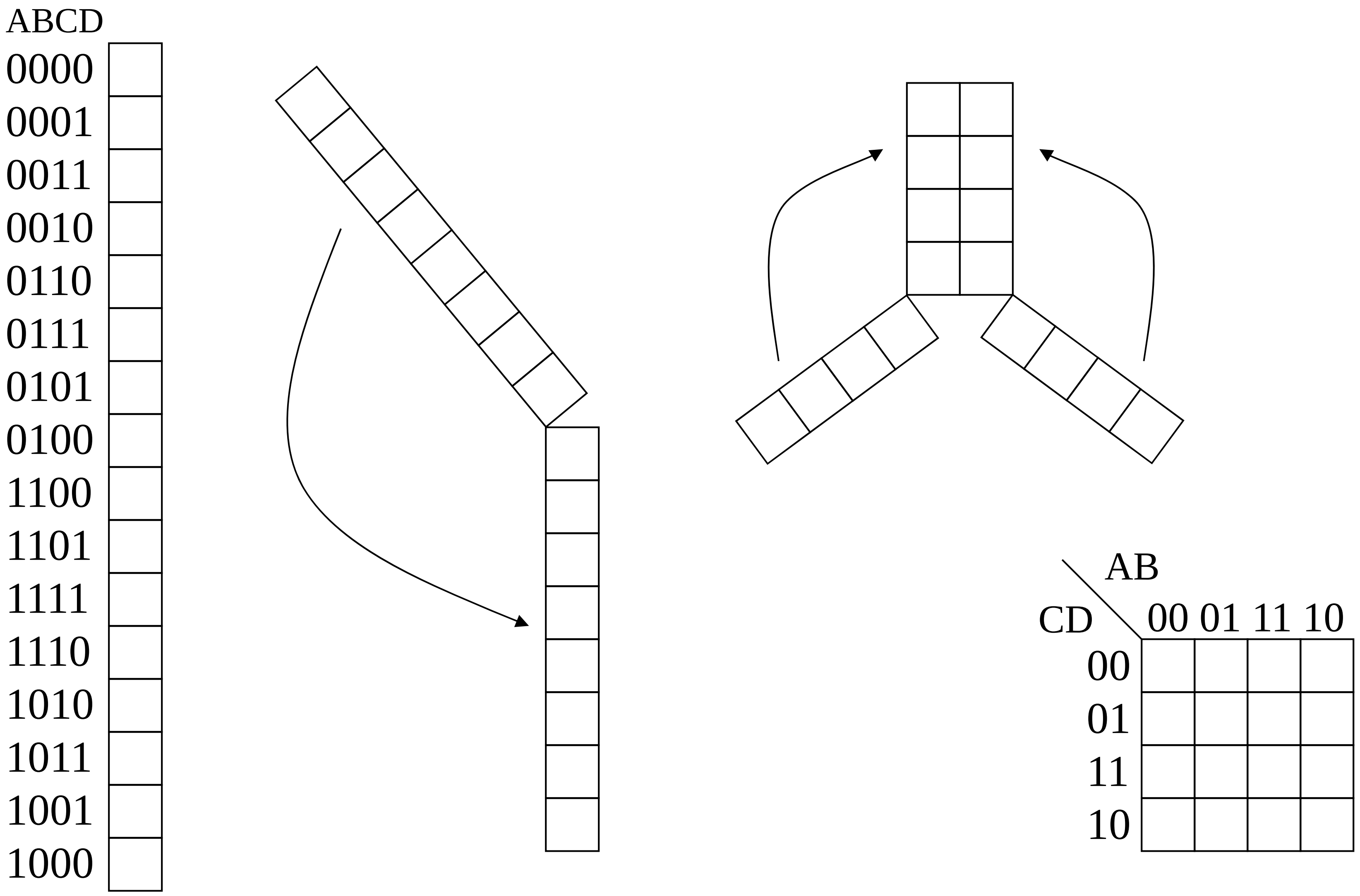
A	B	C	D	minterm
0	0	0	0	m0
0	0	0	1	m1
0	0	1	0	m2
0	0	1	1	m3
0	1	0	0	m4
0	1	0	1	m5
0	1	1	0	m6
0	1	1	1	m7
1	0	0	0	m8
1	0	0	1	m9
1	0	1	0	m10
1	0	1	1	m11
1	1	0	0	m12
1	1	0	1	m13
1	1	1	0	m14
1	1	1	1	m15

Key is to use gray code for row order

A	B	C	D	minterm
0	0	0	0	m0
0	0	0	1	m1
0	0	1	1	m3
0	0	1	0	m2
0	1	1	0	m6
0	1	1	1	m7
0	1	0	1	m5
0	1	0	0	m4
1	1	0	0	m12
1	1	0	1	m13
1	1	1	1	m15
1	1	1	0	m14
1	0	1	0	m10
1	0	1	1	m11
1	0	0	1	m9
1	0	0	0	m8

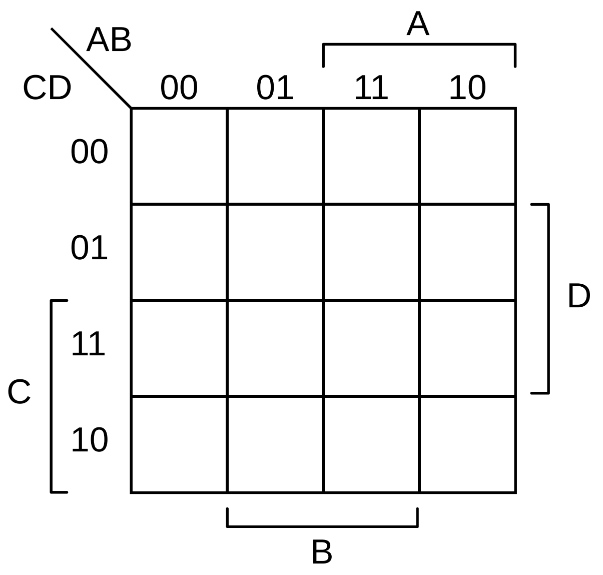
This helps but it's still hard to see all possible adjacencies.

Folding of Gray Code Truth Table into K-map



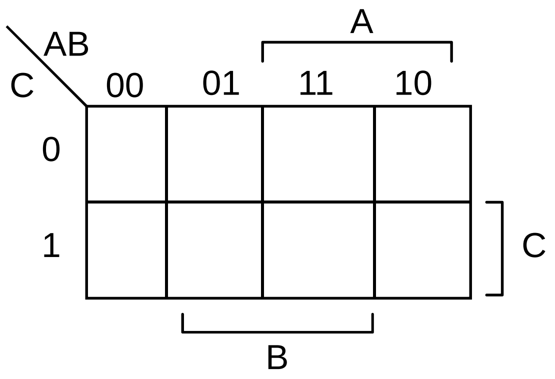
K-map Minimization (cont.)

- For any cell in 2-D array, there are four direct neighbors (top, bottom, left, right)
- 2-D array can therefore show adjacencies of up to four variables.



Four variable K-map

Three variable K-map

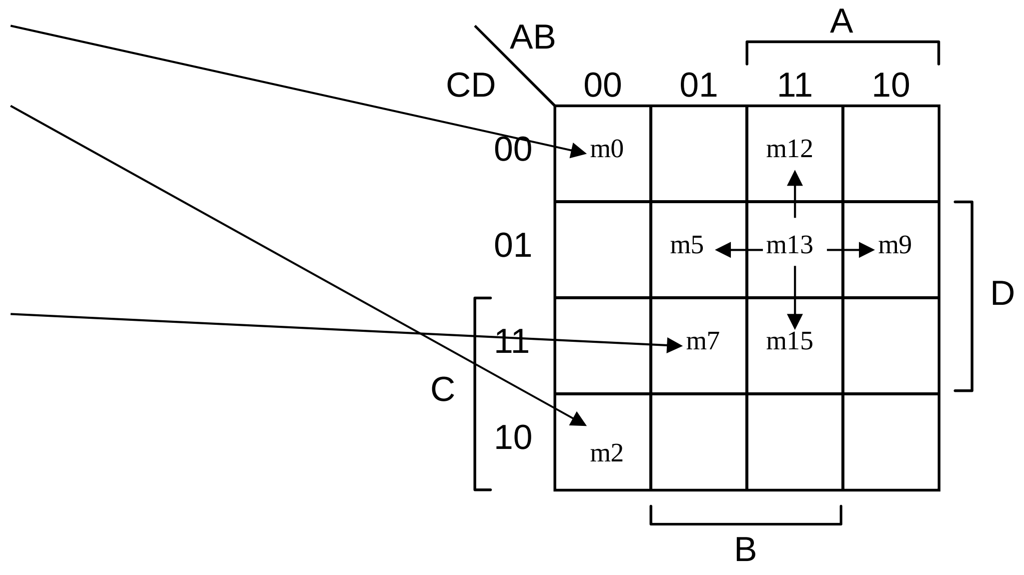


Don’t forget that cells are adjacent top to bottom and side to side.

Truth Table to K-map

Number of TT rows MUST match number of K-map cells

A	B	C	D	F
0	0	0	0	x
0	0	0	1	x
0	0	1	0	x
0	0	1	1	x
0	1	0	0	x
0	1	0	1	x
0	1	1	0	x
0	1	1	1	x
1	0	0	0	x
1	0	0	1	x
1	0	1	0	x
1	0	1	1	x
1	1	0	0	x
1	1	0	1	x
1	1	1	0	x
1	1	1	1	x



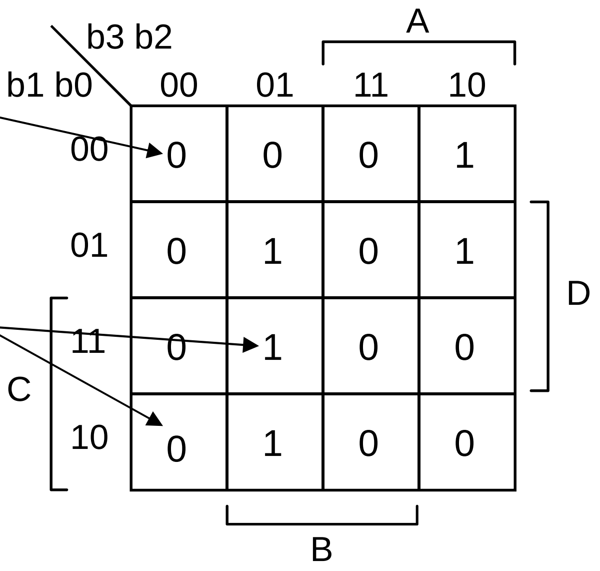
Note different ways K-map is labeled

K-map Minimization of X_3

Entry of TT data into K-map

b3	b2	b1	b0	x3
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	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	-

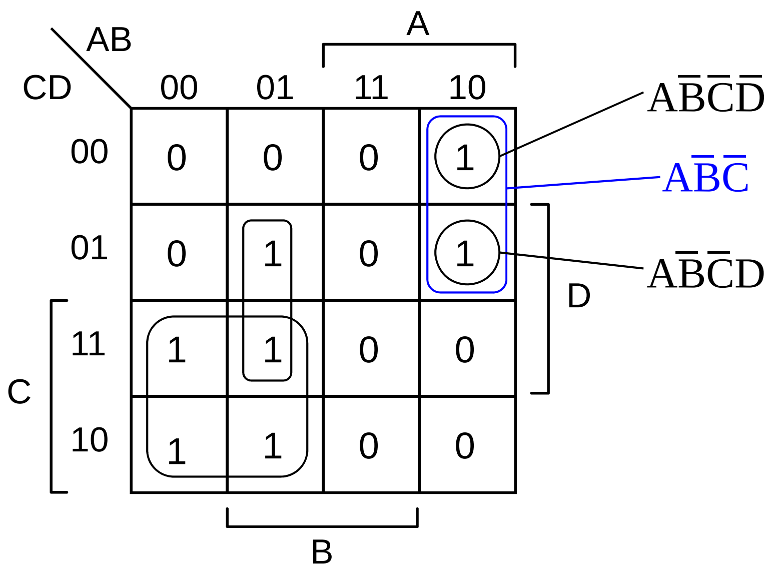
Use 0's for now



b1 b0 \ b3 b2	00	01	11	10
00	0	0	0	1
01	0	1	0	1
11	0	1	0	0
10	0	1	0	0

Watch out for ordering of 10 and 11 rows and columns!

Grouping - Applying Adjacency



If two cells have the same value and are next to each other, the terms are adjacent.

This adjacency is shown by enclosing them.

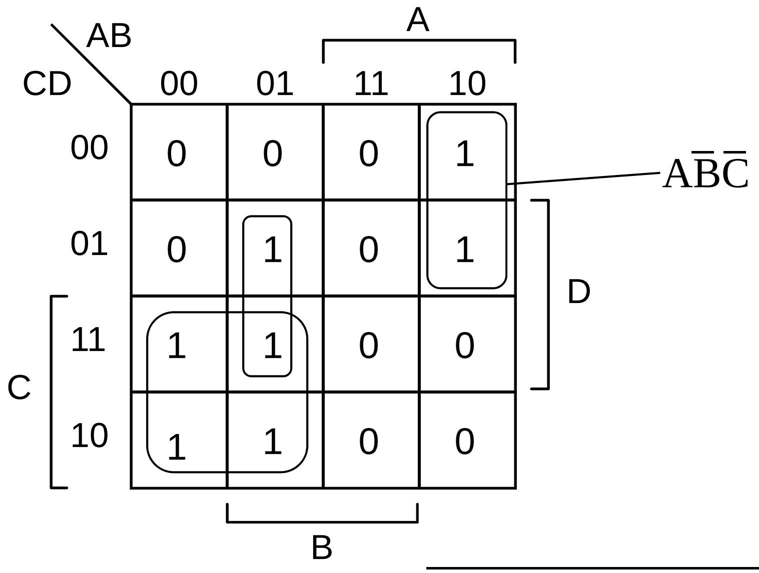
Groups can have common cells.

Group size is a power of 2 and groups are rectangular.

You can group 0s or 1s.

Reading the Groups

If 1s grouped, the expression is a product term, 0s - sum term.



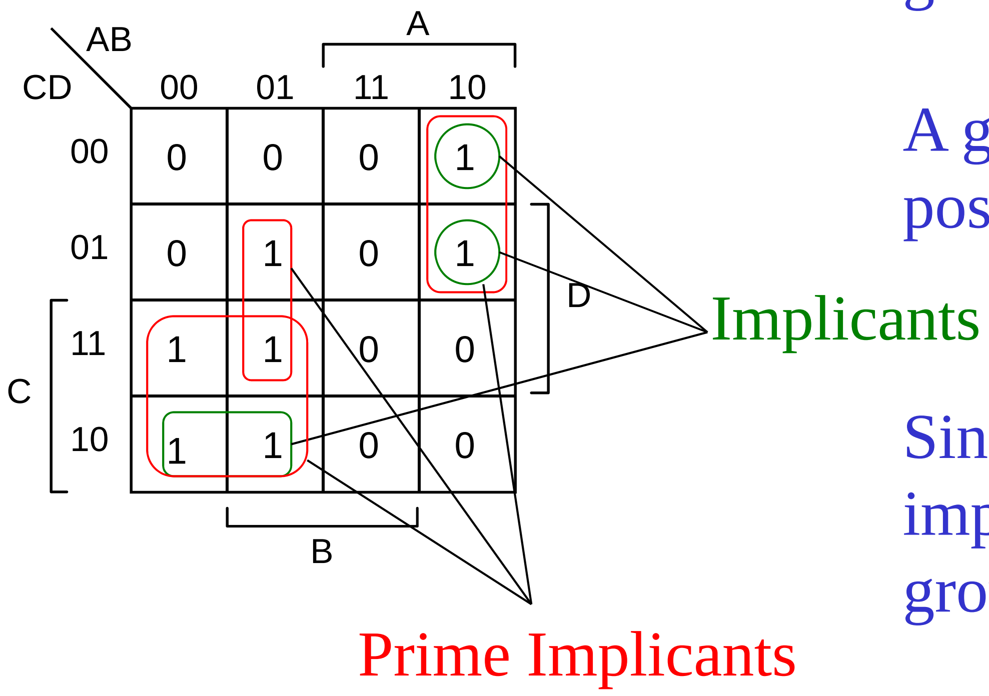
Within group, note when variable values change as you go cell to cell. This determines how the term expression is formed by the following table

	Grouping 1s	Grouping 0s
Variable changes	Exclude	Exclude
Variable constant 0	Inc. comp.	Inc. true
Variable constant 1	Inc. true	Inc. comp.

Reading the Groups (cont.)

- When reading the term expression…
  - If the associated variable value changes within the group, the variable is dropped from the term
  - If reading 1s, a constant 1 value indicates that the associated variable is true in the AND term
  - If reading 0s, a constant 0 value indicates that the associated variable is true in the OR term

Implicants and Prime Implicants



Single cells or groups that could be part of a larger group are know as implicants

A group that is as large as possible is a prime implicant

Single cells can be prime implicants is they cannot be grouped with any other cell

Implicants and Minimal Expressions

- Any set of implicants that encloses (covers) all values is “sufficient”; i.e. the associated logical expression represents the desired function.
  - All minterms or maxterms are sufficient.
- The smallest set of prime implicants that covers all values forms a minimal expression for the desired function.
  - There may be more than one minimal set.

Essential and Secondary Prime Implicants

- If a prime implicant has any cell that is not covered by any other prime implicant, it is an "essential prime implicant"
- If a prime implicant is not essential is is a "secondary prime implicant"
- A minimal set includes ALL essential prime implicants and the minimum number of secondary PIs as needed to cover all values.

K-map Minimization Method

- Technique is valid for either 1s or 0s

    A) Find all prime implicants (largest groups of 1s or 0s in order of largest to smallest)

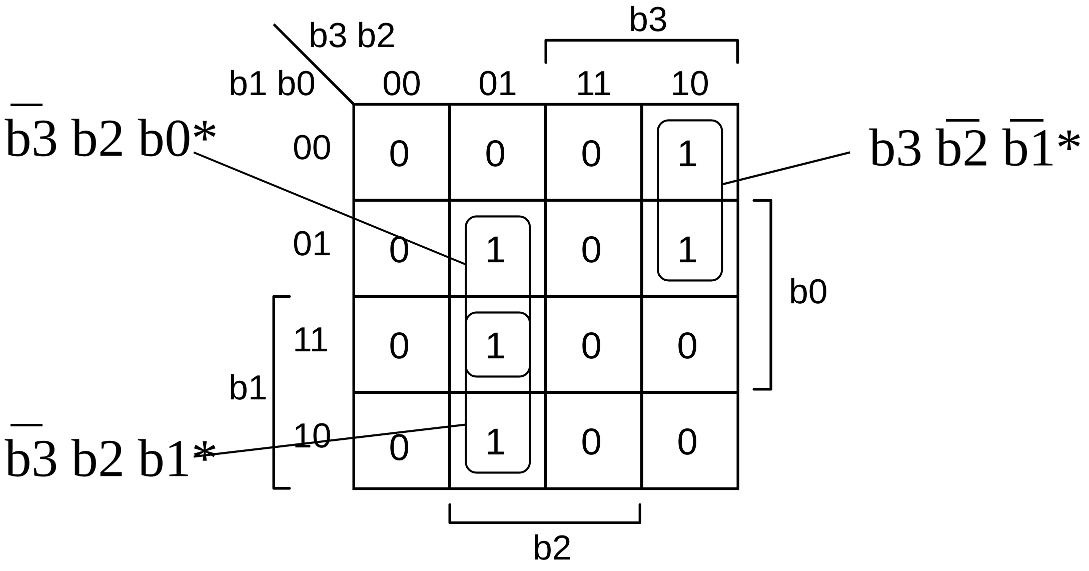
    B) Identify minimal set of PIs

        1) Find all essential PIs

        2) Find smallest set of secondary PIs

The resulting expression is minimal.

K-map Minimization of X_3 (CONT.)



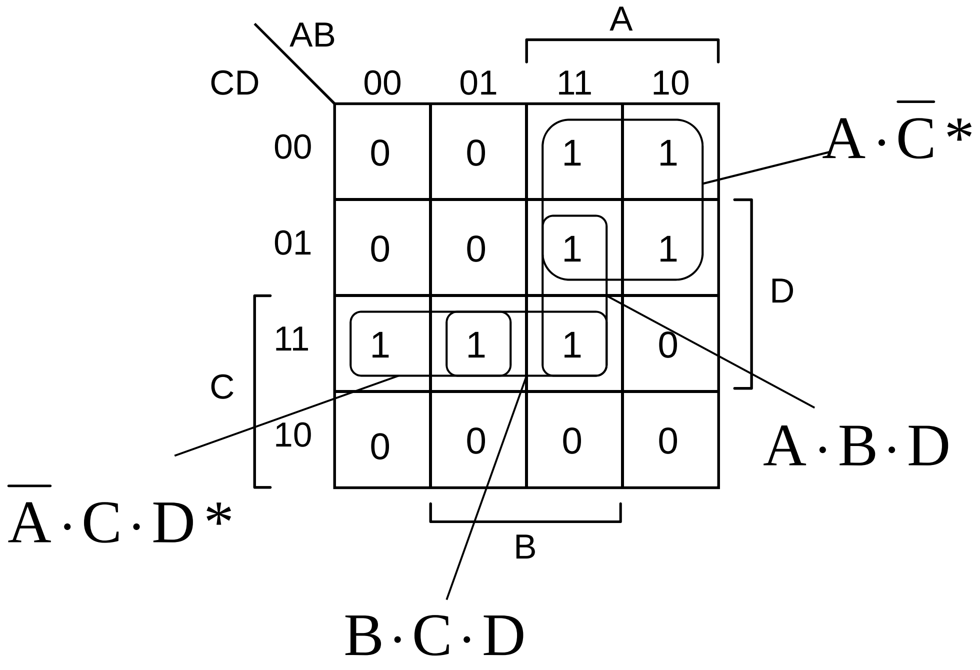
We want a sum of products expression so we circle 1s.

* PIs are essential; no implicants remain (no secondary PIs).

The minimal expression is:

$$X_3 = b3\,\overline{b2}\,\overline{b1} + \overline{b3}\,b2\,b1 + \overline{b3}\,b2\,b0$$

Another K-map Minimization Example

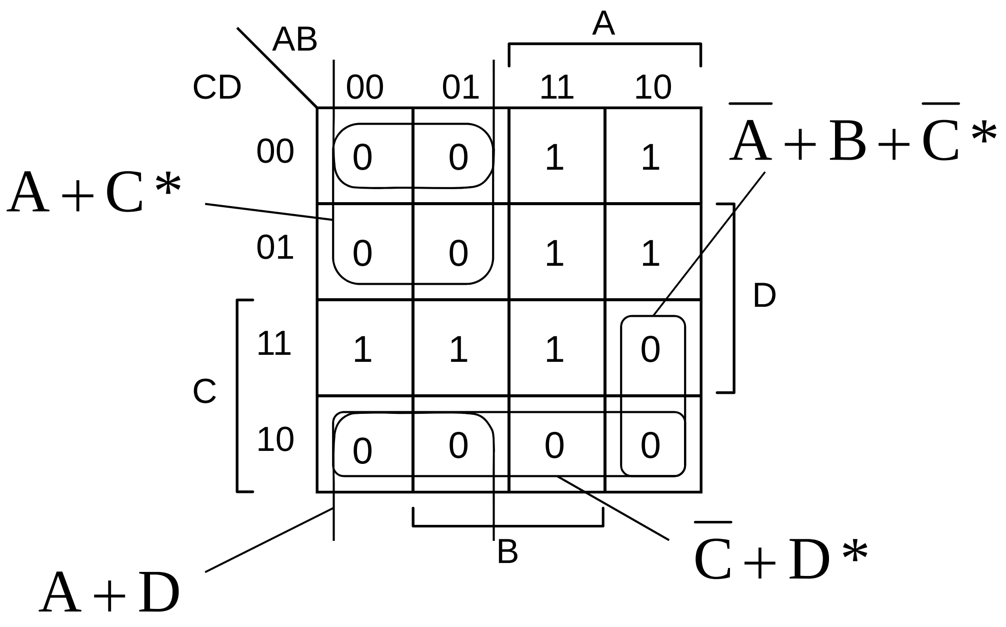


We want a sum of products expression so we circle 1s.
* PIs are essential; and we have 2 secondary PIs. The minimal expressions are:

$$F = A \cdot \overline{C} + \overline{A} \cdot C \cdot D + B \cdot C \cdot D$$

$$F = A \cdot \overline{C} + \overline{A} \cdot C \cdot D + A \cdot B \cdot D$$

A 3rd K-map Minimization Example

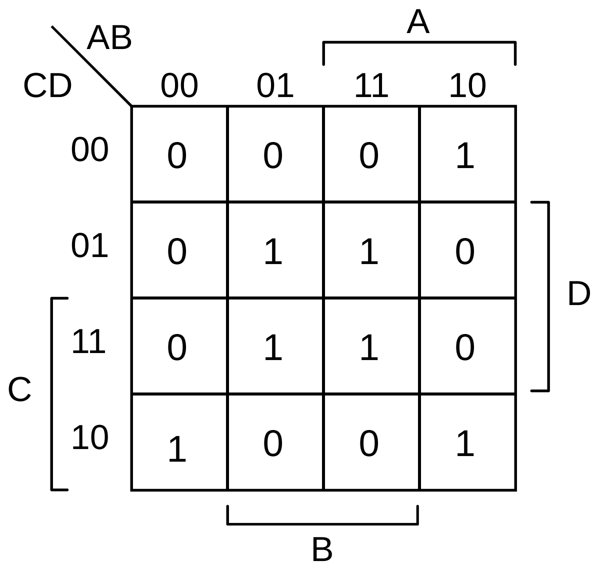


We want a product of sums expression so we circle 0s.
* PIs are essential; and we have 1 secondary PI which is redundant.
The minimal expression is:

$$F=(A+C)\cdot(\overline{C}+D)\cdot(\overline{A}+B+\overline{C})$$

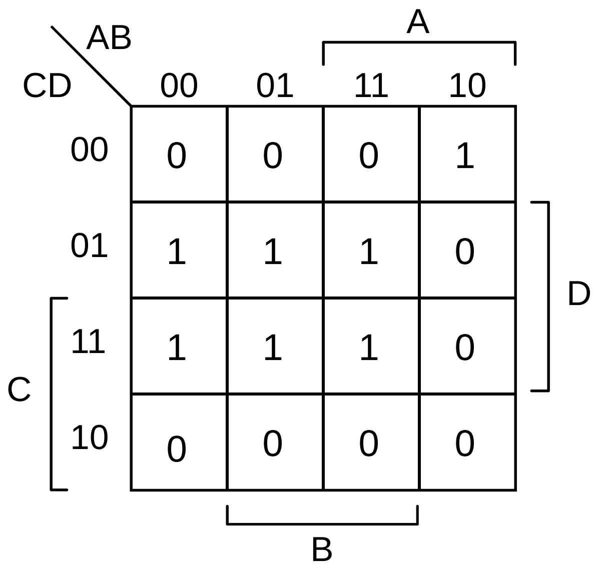
5 Variable K Maps

- Uses two 4 variable maps side-by-side
  - groups spanning both maps occupy the same place in both maps



CD \ AB	00	01	11	10
00	0	0	0	1
01	0	1	1	0
11	0	1	1	0
10	1	0	0	1

E = 0

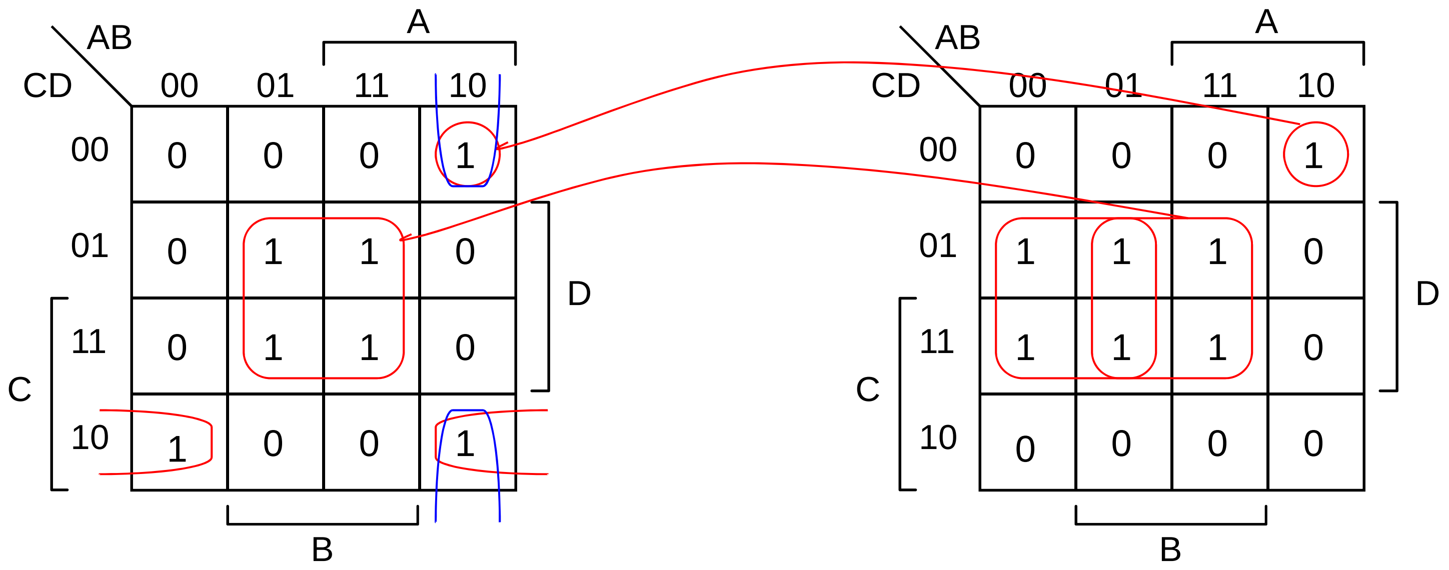


CD \ AB	00	01	11	10
00	0	0	0	1
01	1	1	1	0
11	1	1	1	0
10	0	0	0	0

E = 1

f**(A,B,C,D,E) = Σm(3,4,7,10,11, 14,15,16,17,20,26,27,30 31)**

5 Variable K Maps



E = 0

f **(A,B,C,D,E) = Σm(3,4,7,10,11, 14,15,16,17,20,26,27,30 31)**

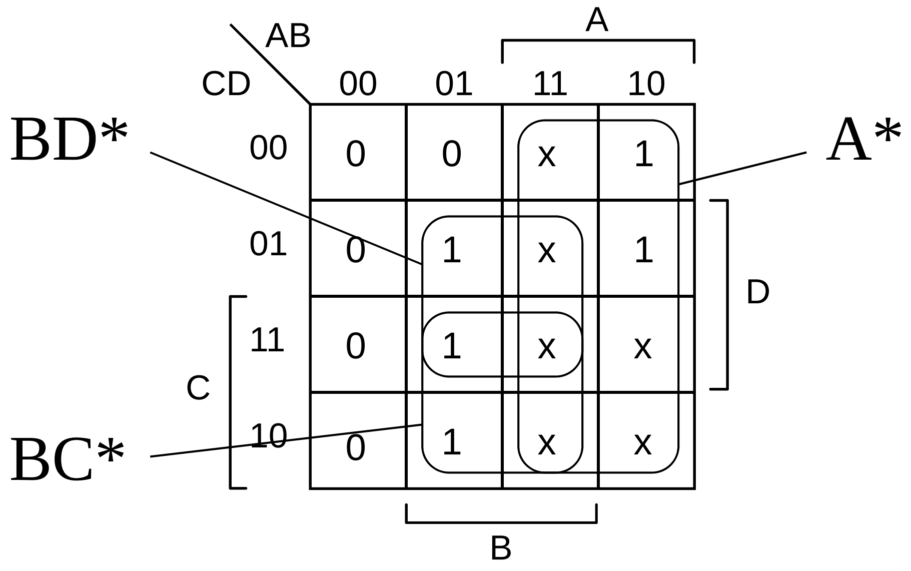
E = 1

$$F(A,B,C,D,E) = B\,D + \overline{A}\,D\,E + A\,\overline{B}\,\overline{C}\,\overline{D} + \overline{B}\,C\,\overline{D}\,\overline{E}$$

Don’t Cares

- For expression minimization, don’t care values (- or x) can be assigned either 0 or 1
  - Hard to use in algebraic simplification; must evaluate all possible combinations
  - K-map minimization easily handles don’t cares
- Basic don’t care rule for K-maps is include the dc (- or x) in group if it helps to form a larger group; else leave it out

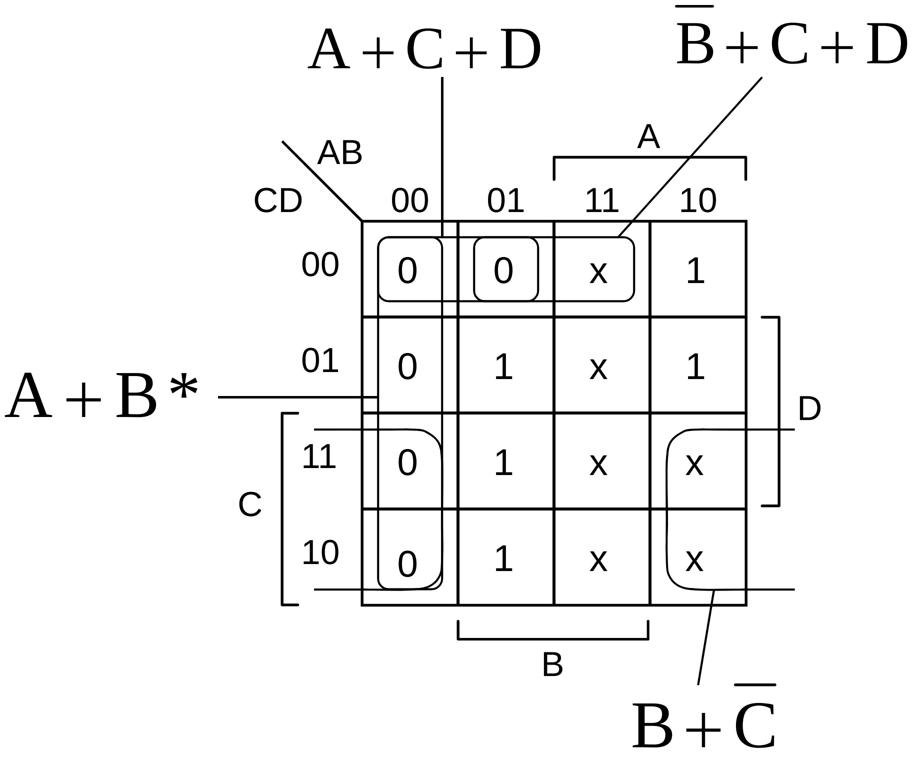
K-map Minimization of X_3 with Don't Cares



We want a sum of products expression so we circle 1s and x's (don't cares)
* PIs are essential; no other implicants remain (no secondary PIs).
The minimal expression is:

$$X_3 = A + BC + BD$$

K-map Minimization of X_3 with Don't Cares



We want a product of sums expression so we circle 0s and x's (don't cares)
* PIs are essential; there are 3 secondary PIs.
The minimal expressions are:

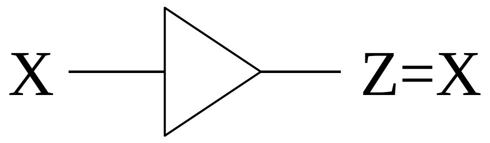
$$F=(A+B)\cdot(\overline{B}+C+D)$$

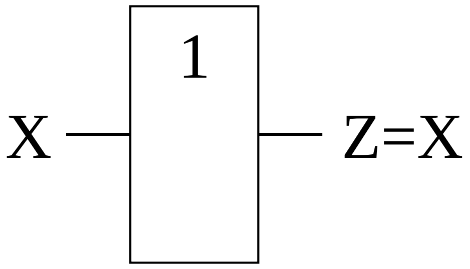
$$F=(A+B)\cdot(A+C+D)$$

Additional Logic Operations

- For two inputs, there are 16 ways we can assign output values
  - Besides AND and OR, five others are useful
- The unary Buffer operation is useful in the real world

X	Z=X
0	0
1	1

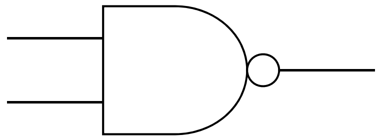


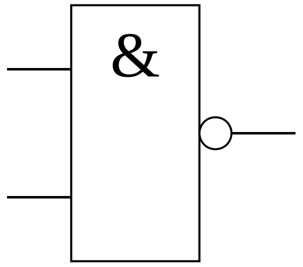


Additional Logic Operations - NAND

- NAND (NOT - AND) is the complement of the AND operation

X	Y	$\overline{X \cdot Y}$
0	0	1
0	1	1
1	0	1
1	1	0

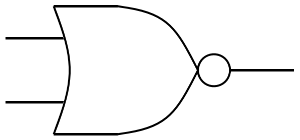


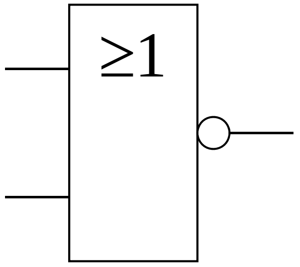


Additional Logic Operations - NOR

- NOR (NOT - OR) is the complement of the OR operation

X	Y	$\overline{X+Y}$
0	0	1
0	1	0
1	0	0
1	1	0

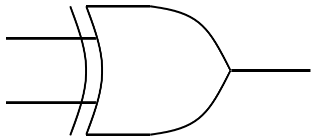


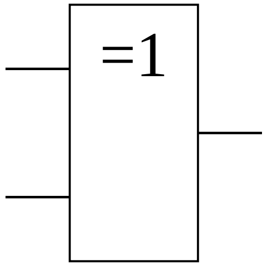


Additional Logic Operations -XOR

- Exclusive OR is similar to the inclusive OR (AKA OR) except output is 0 for 1,1 inputs
- Alternatively the output is 1 when modulo 2 input sum is equal to 1

X	Y	$X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0

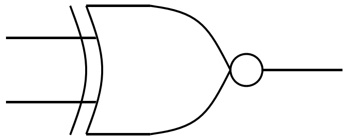


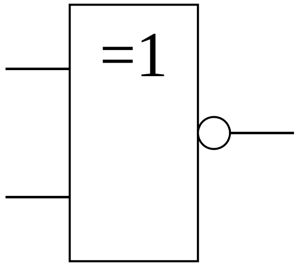


Additional Logic Operations - XNOR

- Exclusive NOR is the complement of the XOR operation
- Alternatively the output is 1 when modulo 2 input sum is not equal to 1

X	Y	$\overline{X \oplus Y}$
0	0	1
0	1	0
1	0	0
1	1	1

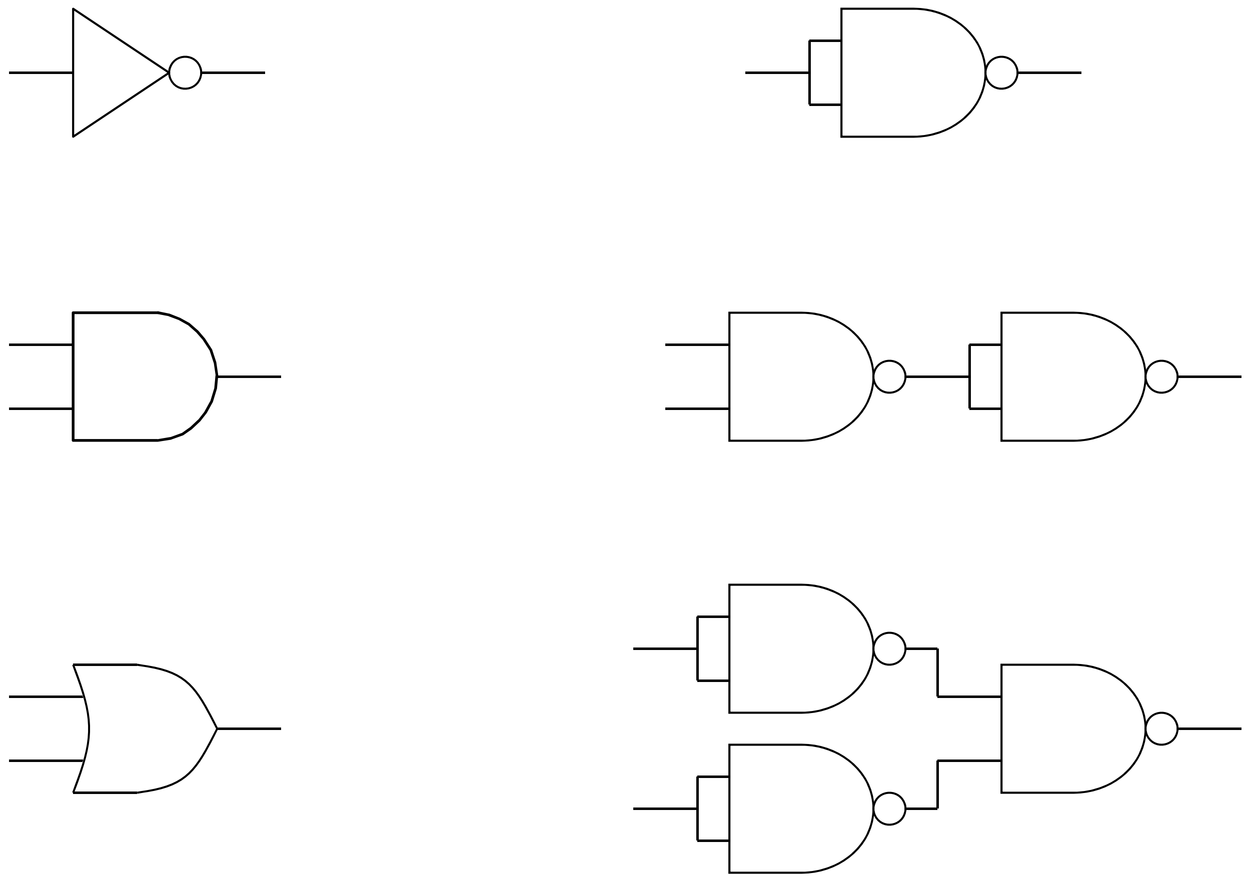




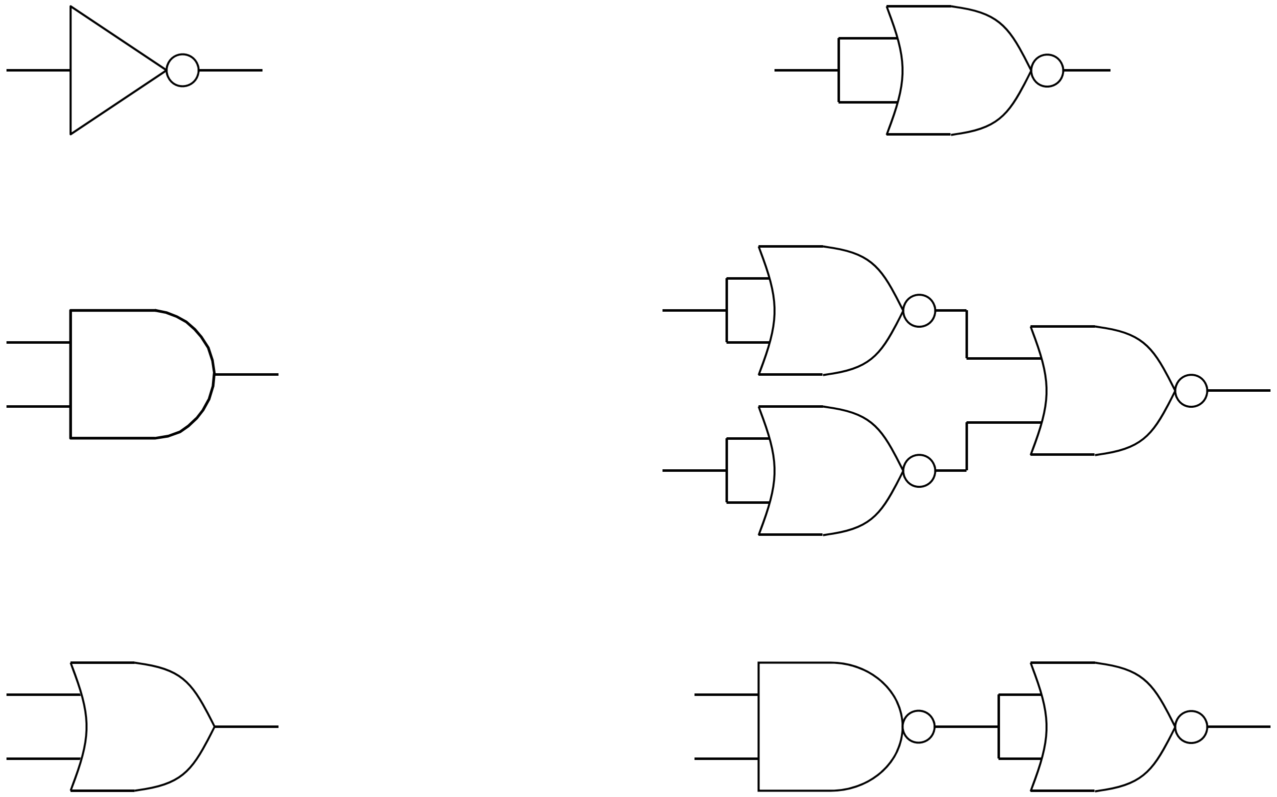
Minimal Logic Operator Sets

- AND , OR, NOT are all that's needed to express any combinational logic function as switching algebra expression
  - operators are all that were originally defined
- Two other minimal logic operator sets exist
  - Just NAND gates
  - Just NOR gates
- We can demonstrate how just NANDs or NORs can do AND, OR, NOT operations

NAND as a Minimal Set



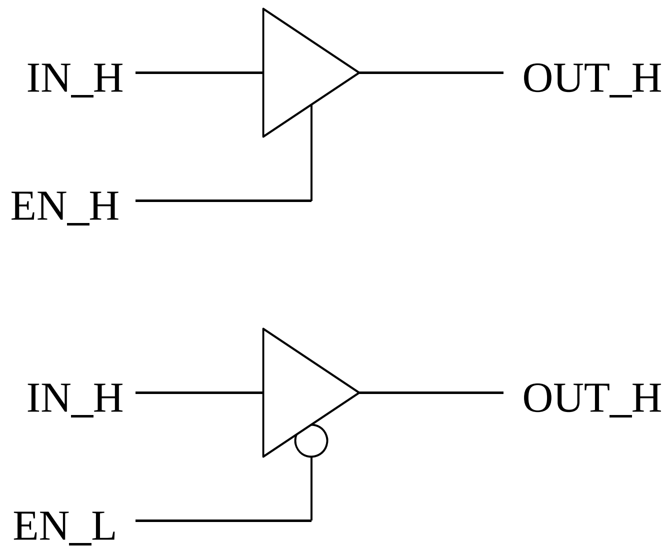
NOR as a Minimal Set

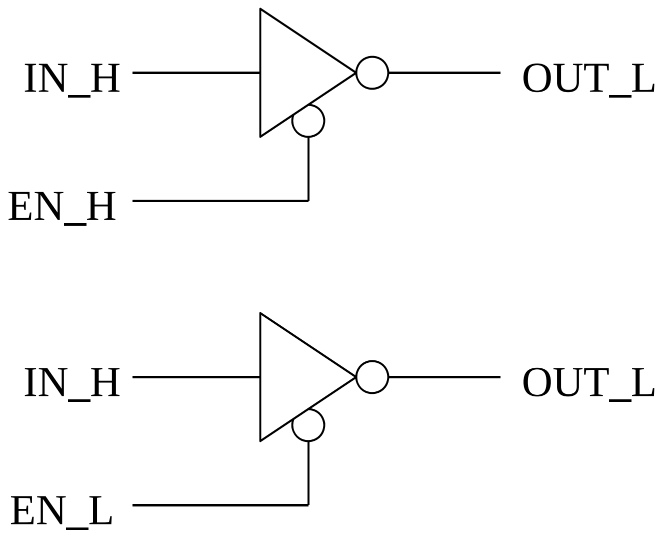


Three State Outputs

- Standard logic gate outputs only have two states; high and low
  - Outputs are effectively either connected to +V or ground (low impedance)
- Certain applications require a logic output that we can “turn off” or disable
  - Output is disconnected (high impedance)
- This is the three-state output
  - May be stand-alone (a buffer) or part of another function output

Three State Buffers





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

Binary Adders and Subtractors

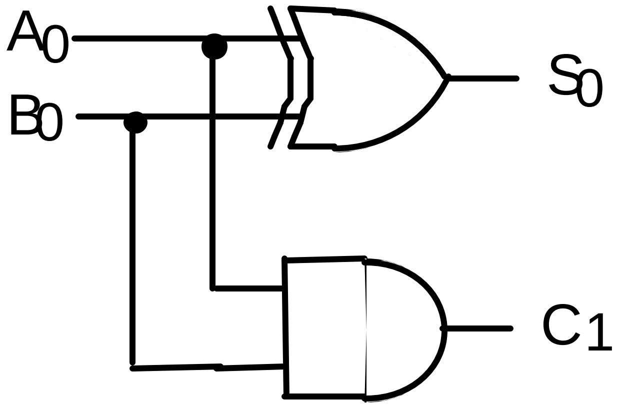
Overview

- Addition and subtraction of binary data is fundamental
  - Need to determine hardware implementation
- Represent inputs and outputs
  - Inputs: single bit values, carry in
  - Outputs: Sum, Carry
- Hardware features
  - Create a single-bit adder and chain together
- Same hardware can be used for addition and subtraction with minor changes
- Dealing with overflow
  - What happens if numbers are too big?

Half Adder

- Add two binary numbers
  - A_0 , B_0 -> single bit inputs
  - S_0 -> single bit sum
  - C_1 -> carry out

A_0	B_0	S_0	C_1
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



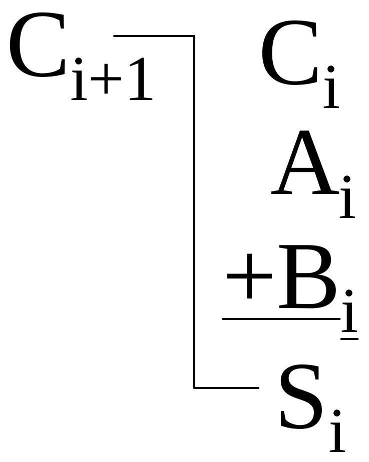
Dec	Binary
1	1
+1	+1
2	10

Multiple-bit Addition

- **Consider single-bit adder for each bit position.**

$$\begin{array}{lcccc} & A_3 & A_2 & A_1 & A_0 \\ A & 0 & 1 & 0 & 1 \end{array} \qquad \begin{array}{lcccc} & B_3 & B_2 & B_1 & B_0 \\ B & 0 & 1 & 1 & 1 \end{array}$$

$$\begin{array}{lcccc} & 1 & 1 & 1 & \\ A & 0 & 1 & 0 & 1 \\ B & 0 & 1 & 1 & 1 \\ \hline & 1 & 1 & 0 & 0 \end{array}$$



Each bit position creates a sum and carry

Full Adder

- Full adder includes carry in C_i
- Notice interesting pattern in Karnaugh map.

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

C_i \ A_iB_i	00	01	11	10
0		1		1
1	1		1	

S_i

Full Adder

- Full adder includes carry in C_i
- Alternative to XOR implementation

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

$$\begin{aligned} S_i = {} & !C_i \ \& \ !A_i \ \& \ B_i \\ & \# \ !C_i \ \& \ A_i \ \& \ !B_i \\ & \# \ C_i \ \& \ !A_i \ \& \ !B_i \\ & \# \ C_i \ \& \ A_i \ \& \ B_i \end{aligned}$$

Full Adder

- **Reduce and/or representations into XORs**

```
Si = !Ci & !Ai & Bi
   # !Ci & Ai & !Bi
   # Ci & !Ai & !Bi
   # Ci & Ai & Bi

Si = !Ci & (!Ai & Bi # Ai & !Bi)
   # Ci & (!Ai & !Bi # Ai & Bi)

Si = !Ci & (Ai $ Bi)
   # Ci & !(Ai $ Bi)

Si = Ci $ (Ai $ Bi)
```

Full Adder

- **Now consider implementation of carry out**
- **Two outputs per full adder bit** (C_{i+1}, S_i)

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

C_i \ A_iB_i	00	01	11	10
0			1	
1		1	1	1

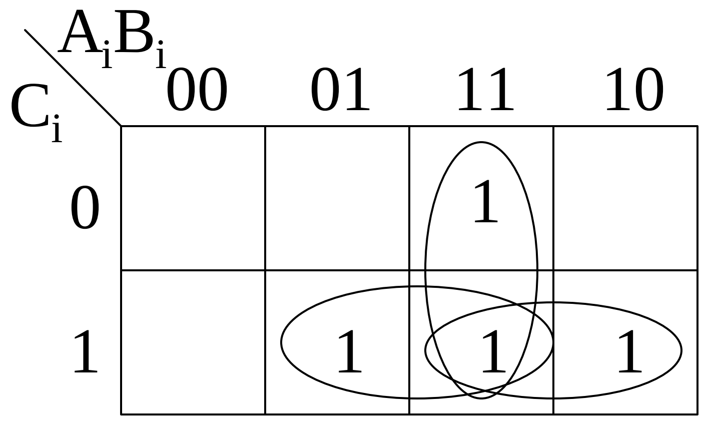
C_{i+1}

Note: 3 inputs

Full Adder

- Now consider implementation of carry out
- Minimize circuit for carry out - C_{i+1}

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



C_{i+1}

```
C_{i+1} = A_i & B_i
        # C_i & B_i
        # C_i & A_i
```

Full Adder

$$
\begin{aligned}
C_{i+1} = {} & A_i \ \& \ B_i \\
& \# \ C_i \ !A_i \ \& \ B_i \\
& \# \ C_i \ \& \ A_i \ \& \ !B_i
\end{aligned}
$$

$$
\begin{aligned}
C_{i+1} = {} & A_i \ \& \ B_i \\
& \# \ C_i \ \& \ (!A_i \ \& \ B_i \ \# \ A_i \ \& \ !B_i)
\end{aligned}
$$

$$C_{i+1} = A_i \ \& \ B_i \ \# \ C_i \ \& \ (A_i \ \$ \ B_i)$$

Recall:

$$S_i = C_i \ \$ \ (A_i \ \$ \ B_i)$$

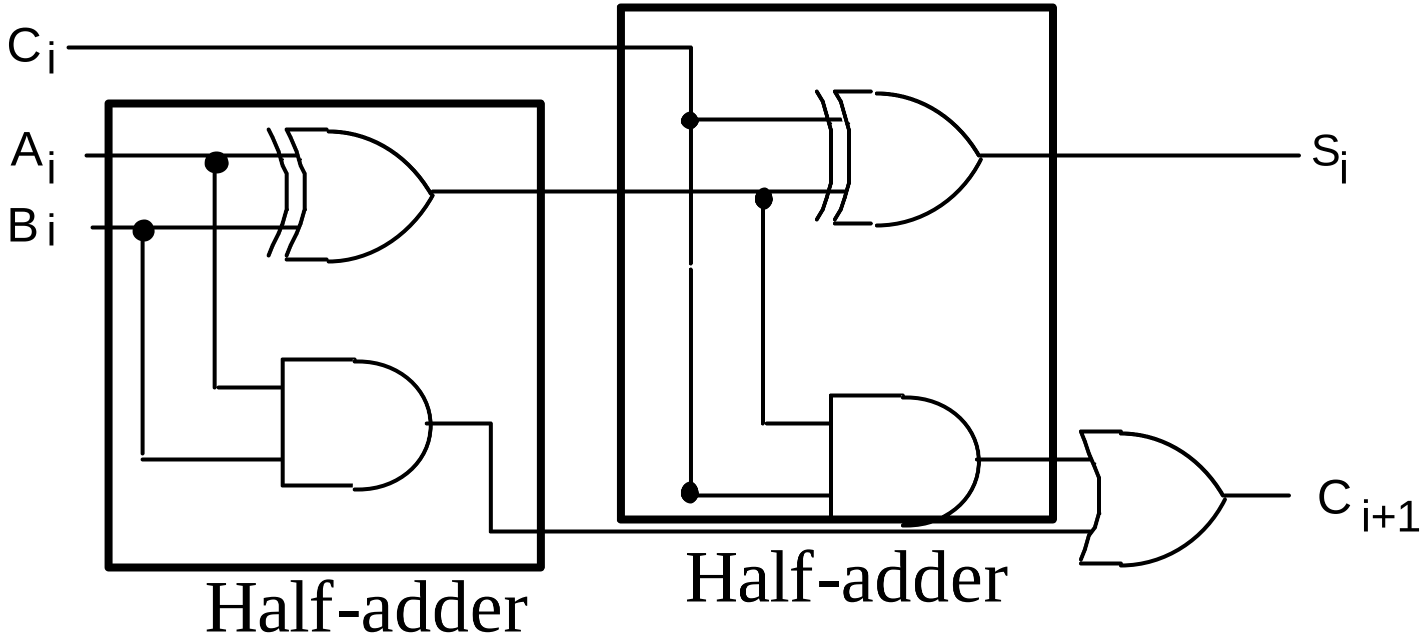
$$C_{i+1} = A_i \ \& \ B_i \ \# \ C_i \ \& \ (A_i \ \$ \ B_i)$$

Full Adder

- Full adder made of several half adders

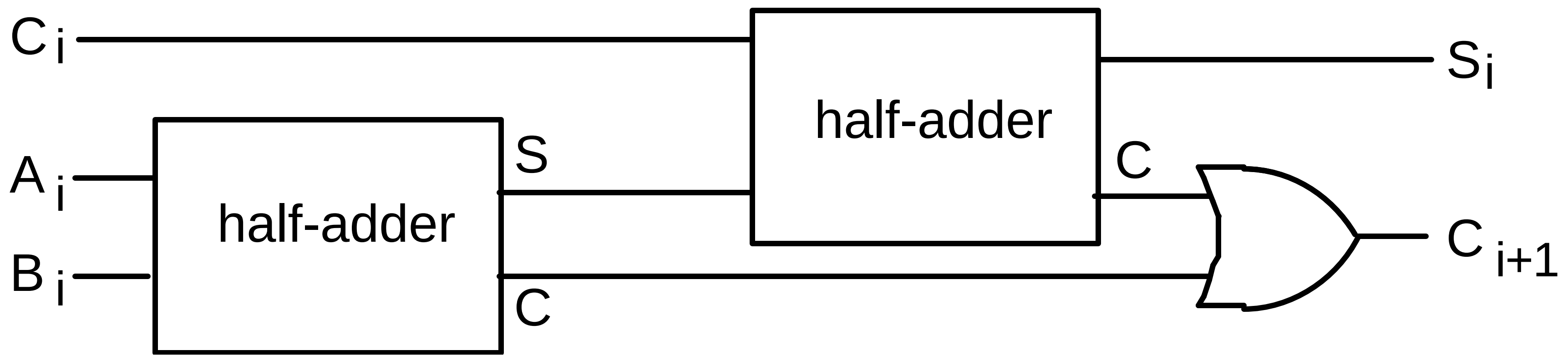
$$S_i = C_i \$ (A_i \$ B_i)$$

$$C_{i+1} = A_i \& B_i \# C_i \& (A_i \$ B_i)$$



Full Adder

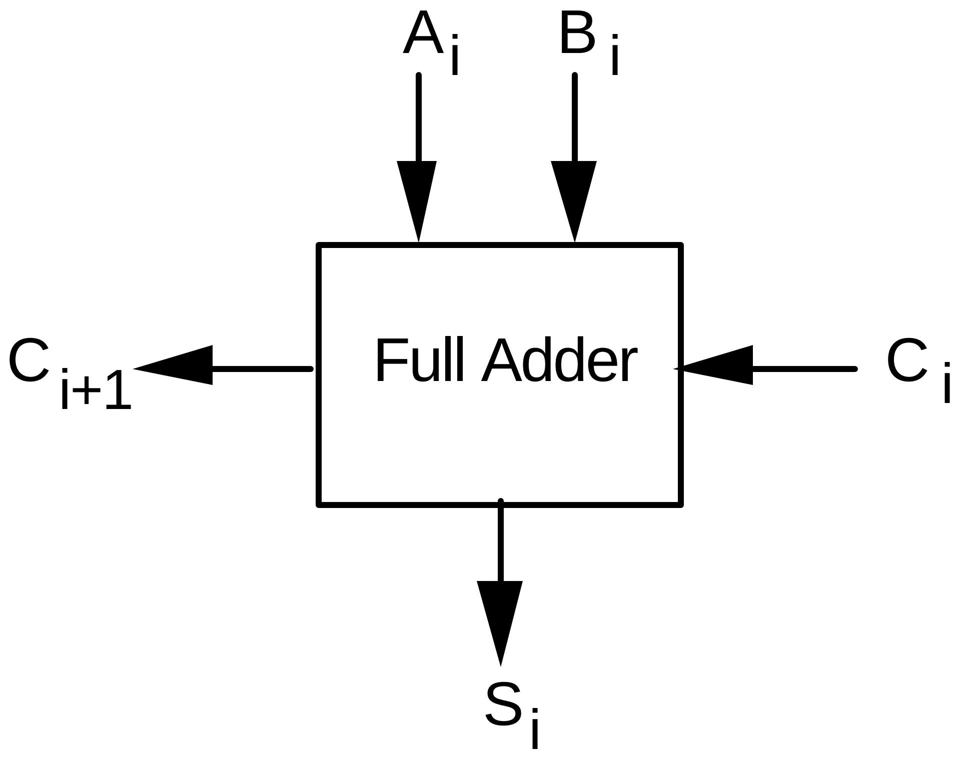
- Hardware repetition simplifies hardware design



A full adder can be made from
two half adders (plus an OR gate).

Full Adder

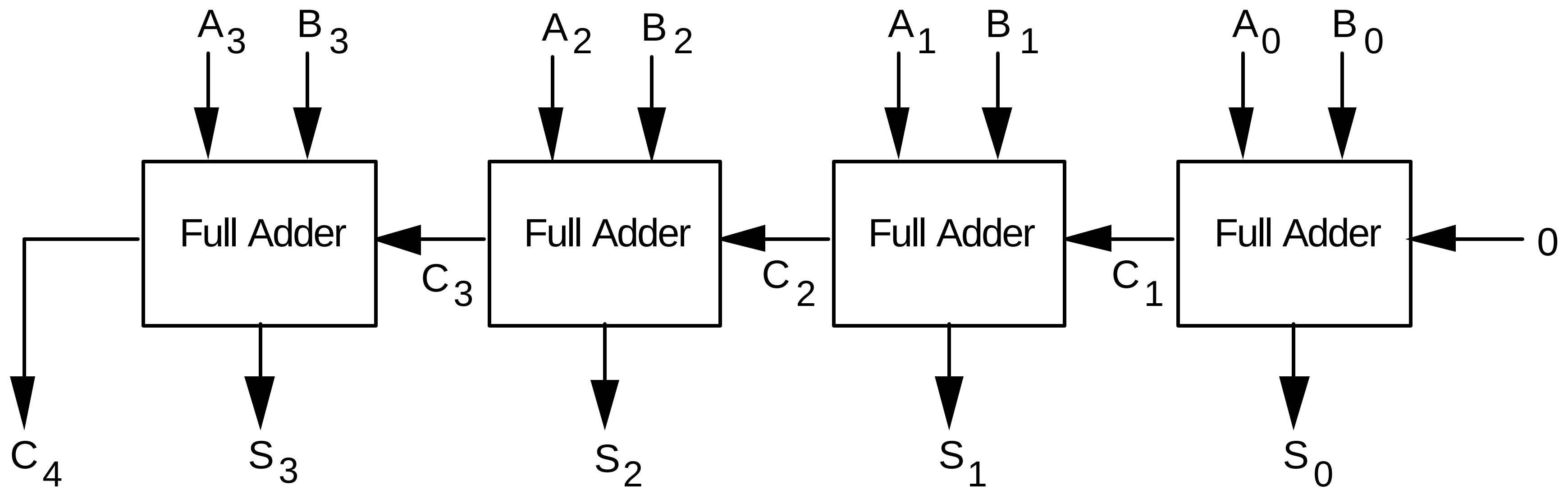
- **Putting it all together**
  - **Single-bit full adder**
  - **Common piece of computer hardware**



Block Diagram

4-Bit Adder

- **Chain single-bit adders together.**
- **What does this do to delay?**



```
C   1 1 1 0
A   0 1 0 1
B   0 1 1 1
    -------
S   1 1 0 0
```

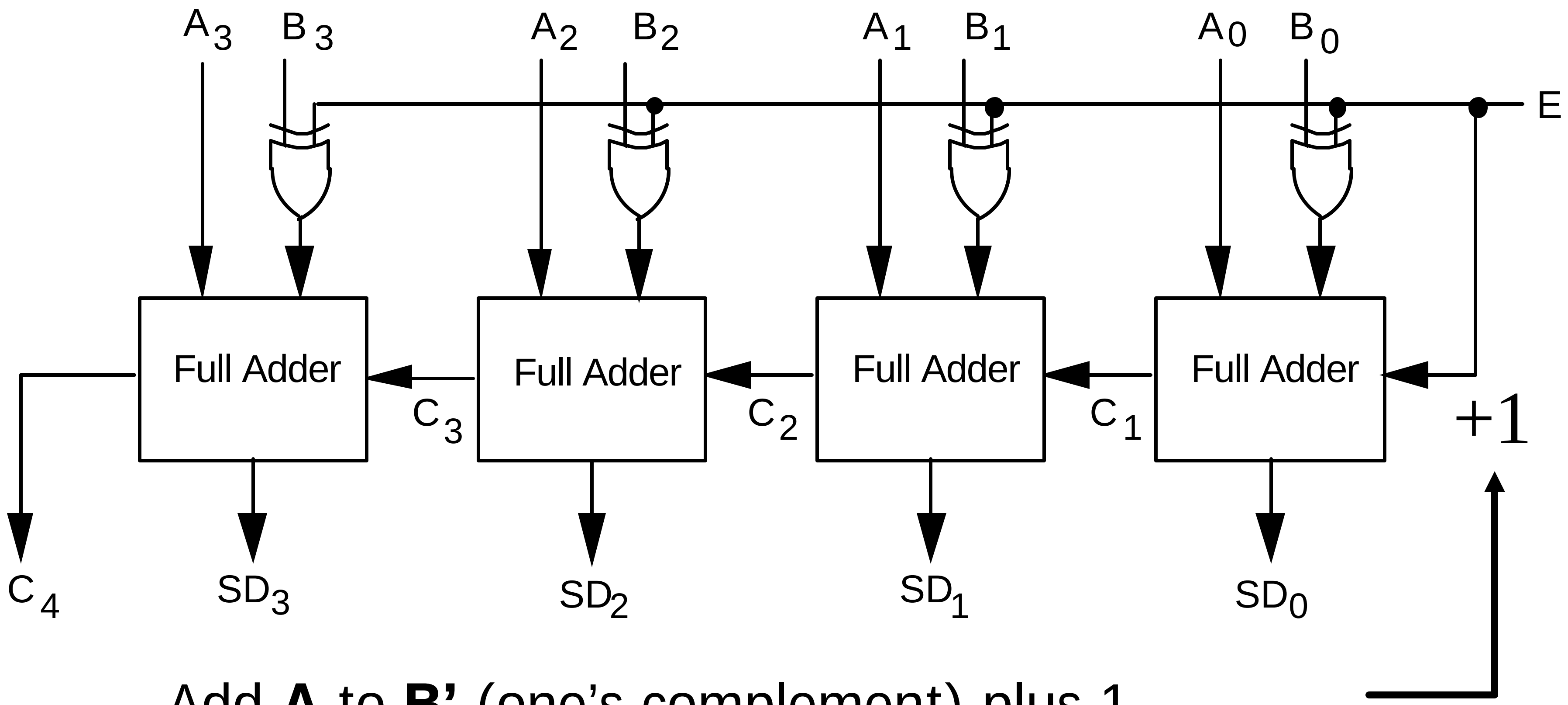
Negative Numbers – 2's Complement.

- Subtracting a number is the same as:
  1. Perform 2's complement
  2. Perform addition
- If we can augment adder with 2's complement hardware?

$$
\begin{array}{rcccc}
1_{10} & = & 01_{16} & = & 00000001 \\
-1_{10} & = & FF_{16} & = & 11111111
\end{array}
$$

$$
\begin{array}{rcccc}
128_{10} & = & 80_{16} & = & 10000000 \\
-128_{10} & = & 80_{16} & = & 10000000
\end{array}
$$

4-bit Subtractor: E = 1

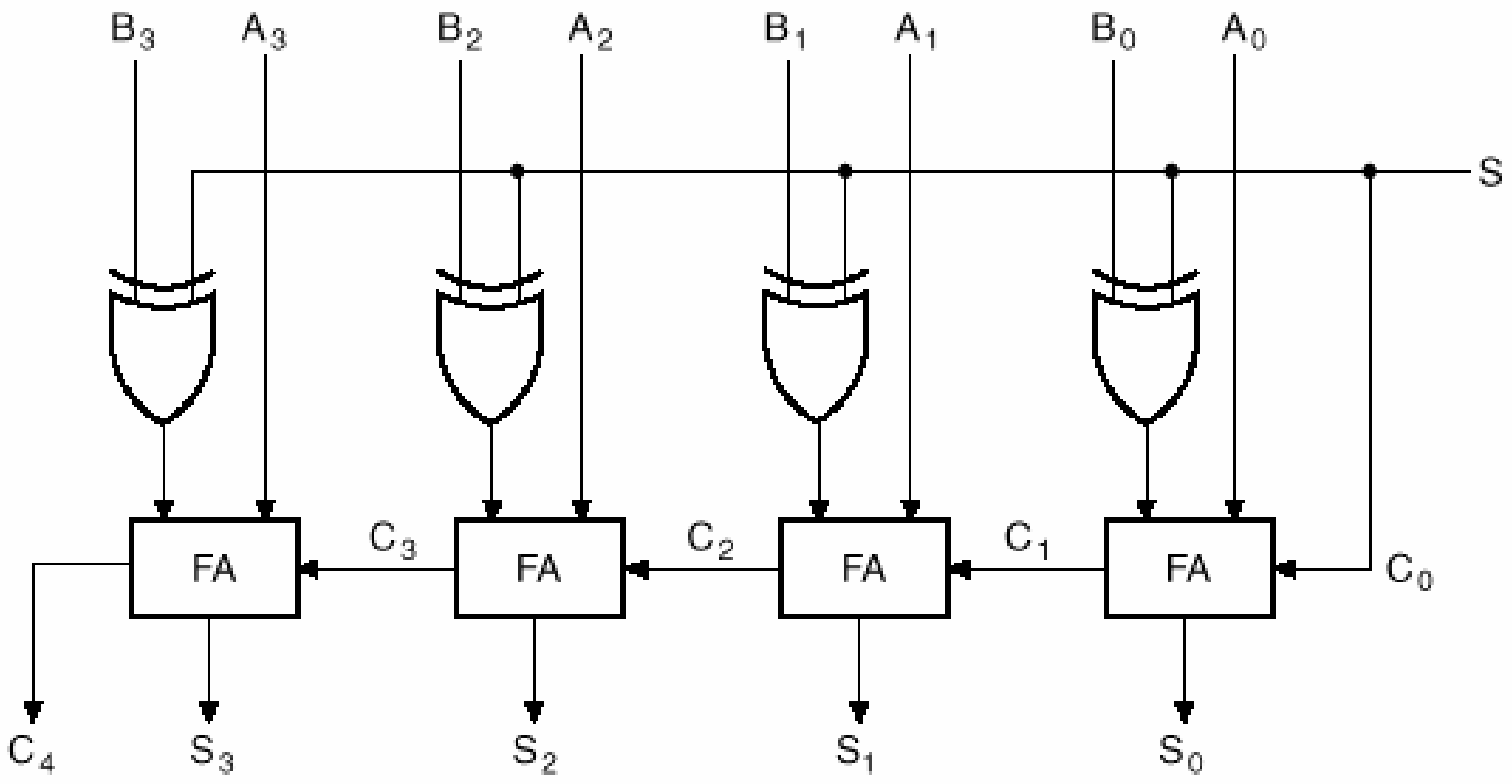


Add **A** to **B’** (one’s complement) plus 1

That is, add **A** to two’s complement of **B**

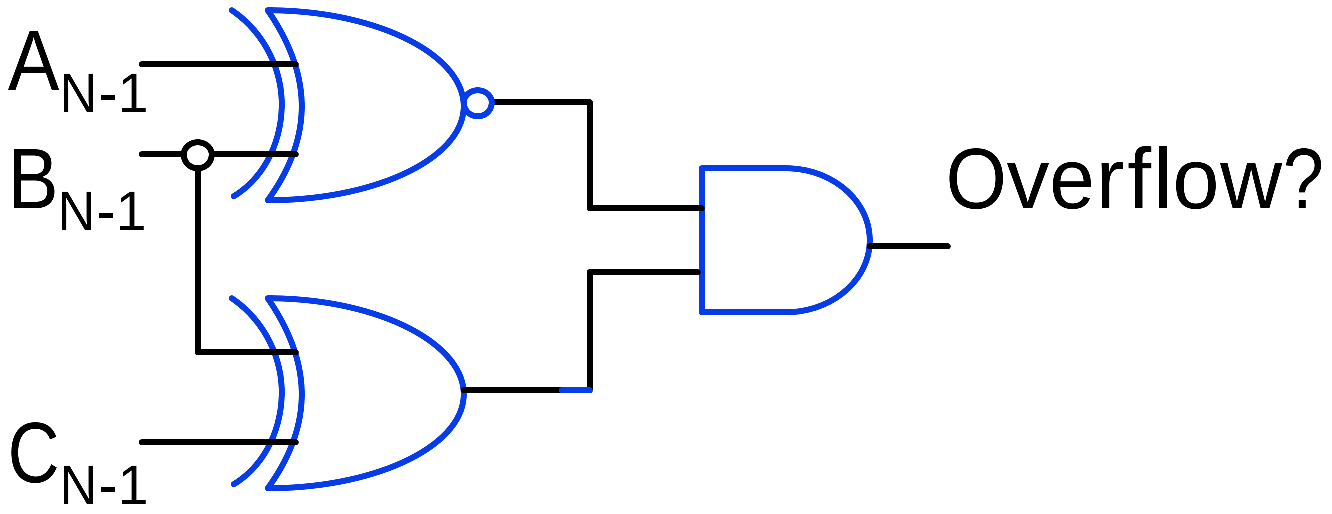
D = **A** - **B**

Adder- Subtractor Circuit



Overflow in two's complement addition

- **Definition: When two values of the same signs are added:**
  - **Result won't fit in the number of bits provided**
  - **Result has the opposite sign.**



Assumes an N-bit adder, with bit N-1 the MSB

Addition cases and overflow

00	01	11	10	00	11
0010	0011	1110	1101	0010	1110
0011	0110	1101	1010	1100	0100
--------	--------	--------	--------	--------	--------
0101	1001	1011	0111	1110	0010
2	3	-2	-3	2	-2
3	6	-3	-6	-4	4
5	-7	-5	7	-2	2
	OFL		OFL		

Summary

- Addition and subtraction are fundamental to computer systems
- Key – create a single bit adder/subtractor
  - Chain the single-bit hardware together to create bigger designs
- The approach is call *ripple-carry* addition
  - Can be slow for large designs
- Overflow is an important issue for computers
  - Processors often have hardware to detect overflow
- Next time: encoders/decoder.

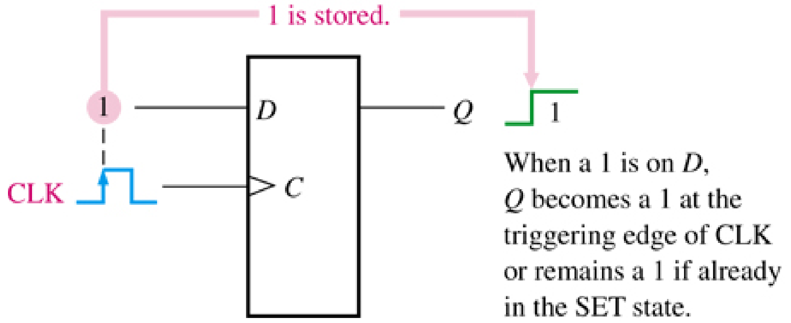
Shift Registers

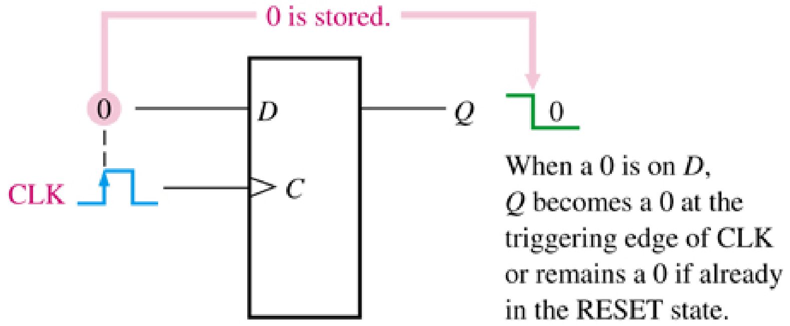
Prof. Young Jin Nam

Basic Functions

- Register *is*
  - A digital circuit which two basic functions: data storage & data movement
  - Consisting of one or more F/Fs used to store & shift data

- Flip-Flop as a Storage Element

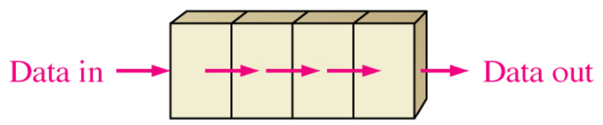

1 is stored.
1
D
Q
1
CLK
C
When a 1 is on D, Q becomes a 1 at the triggering edge of CLK or remains a 1 if already in the SET state.


0 is stored.
0
D
Q
0
CLK
C
When a 0 is on D, Q becomes a 0 at the triggering edge of CLK or remains a 0 if already in the RESET state.

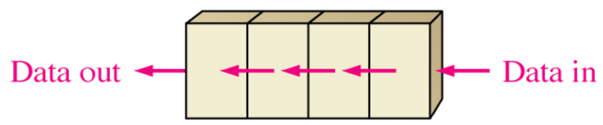
Basic Functions

❑ Basic Data Movement in Shift Registers

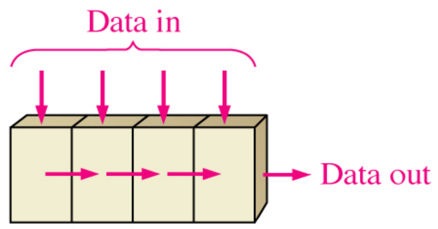
- Shift register can defined by three factors: capacity, the method of data input & output



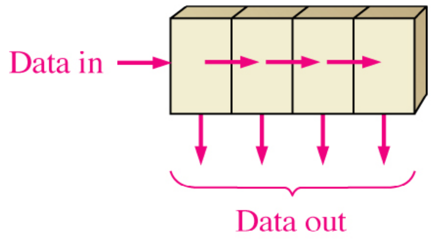
(a) Serial in/shift right/serial out



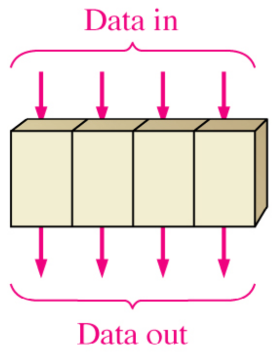
(b) Serial in/shift left/serial out



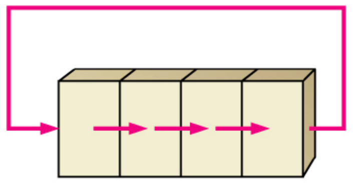
(c) Parallel in/serial out



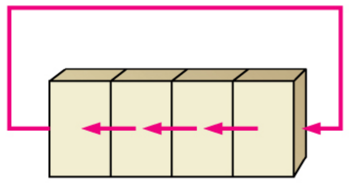
(d) Serial in/parallel out



(e) Parallel in/parallel out



(f) Rotate right



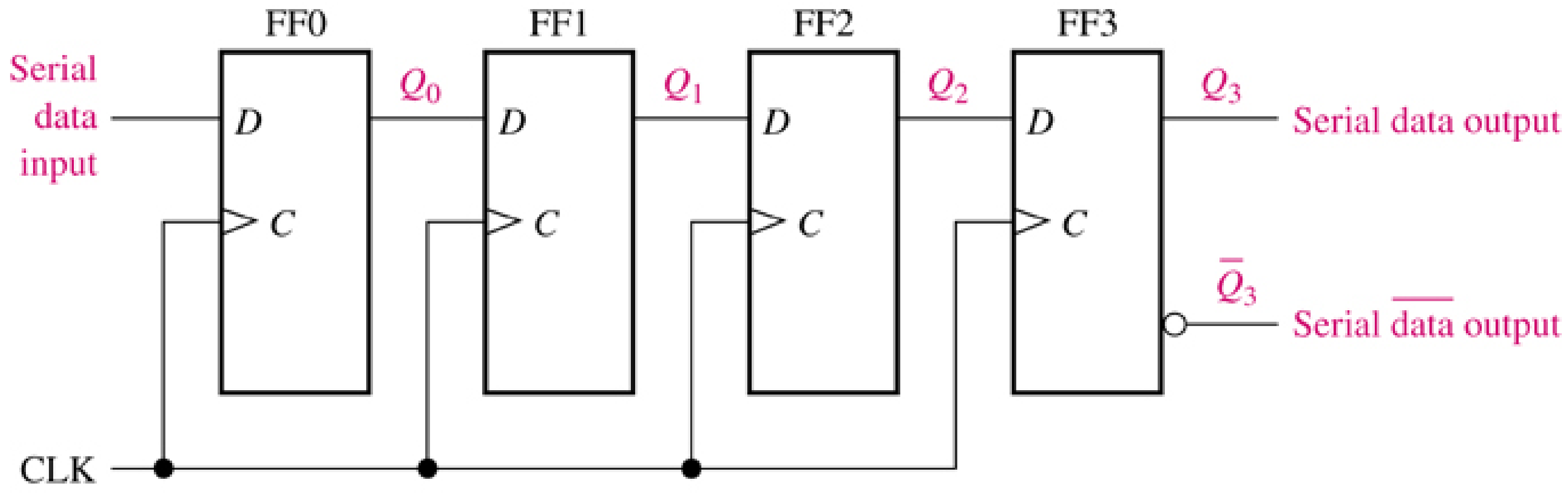
(g) Rotate left

Basic Functions

- Storage Capacity
  - The total # of bits (1s or 0s) of digital data it can retain
  - Each stage(flip-flop) in a shift register represents one bit of storage capacity
  - The # of stages in a register determines its storage capacity

Serial In/Serial Out Shift Register

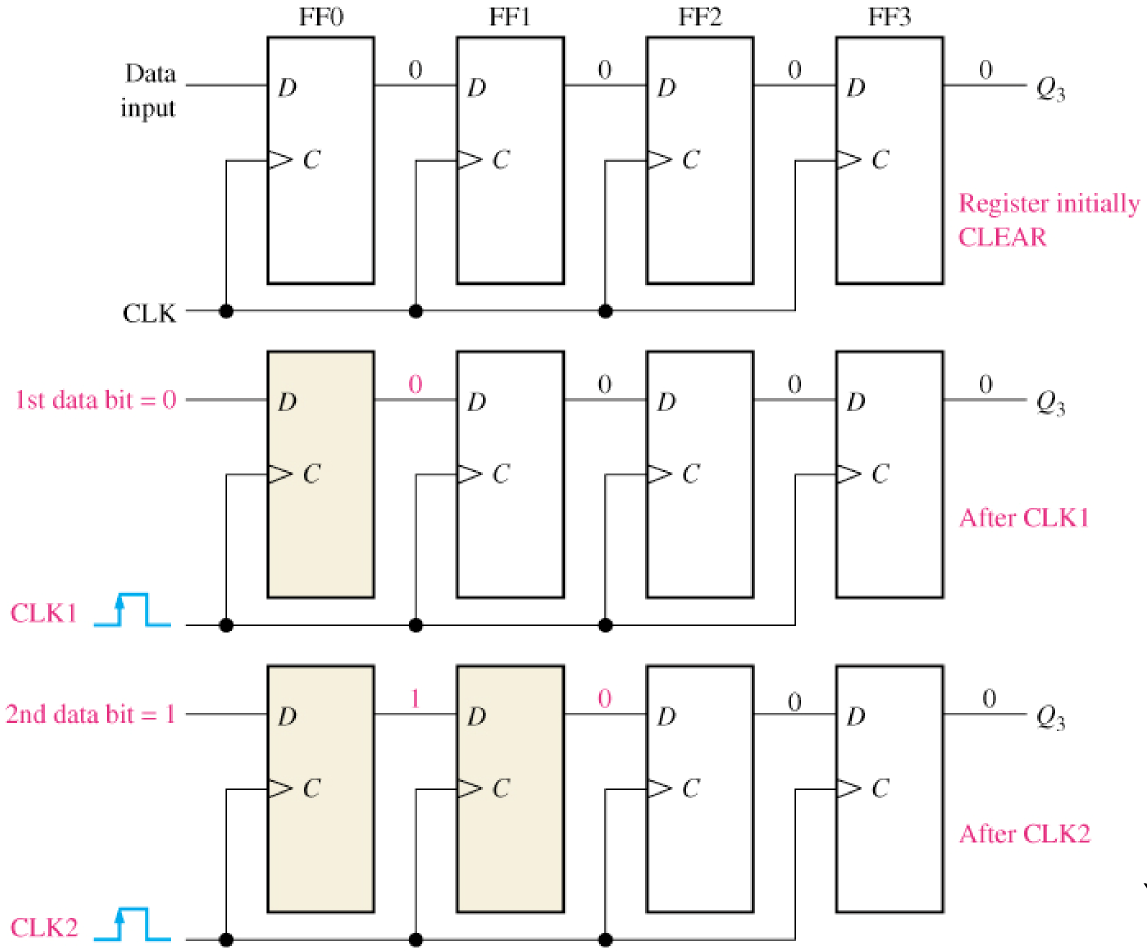
- Serial In/Serial Out Shift Register
  - Accepts data serially (one bit at a time on a single line)
  - Produces the stored information on its output also in serial form



Serial In/Serial Out Shift Register

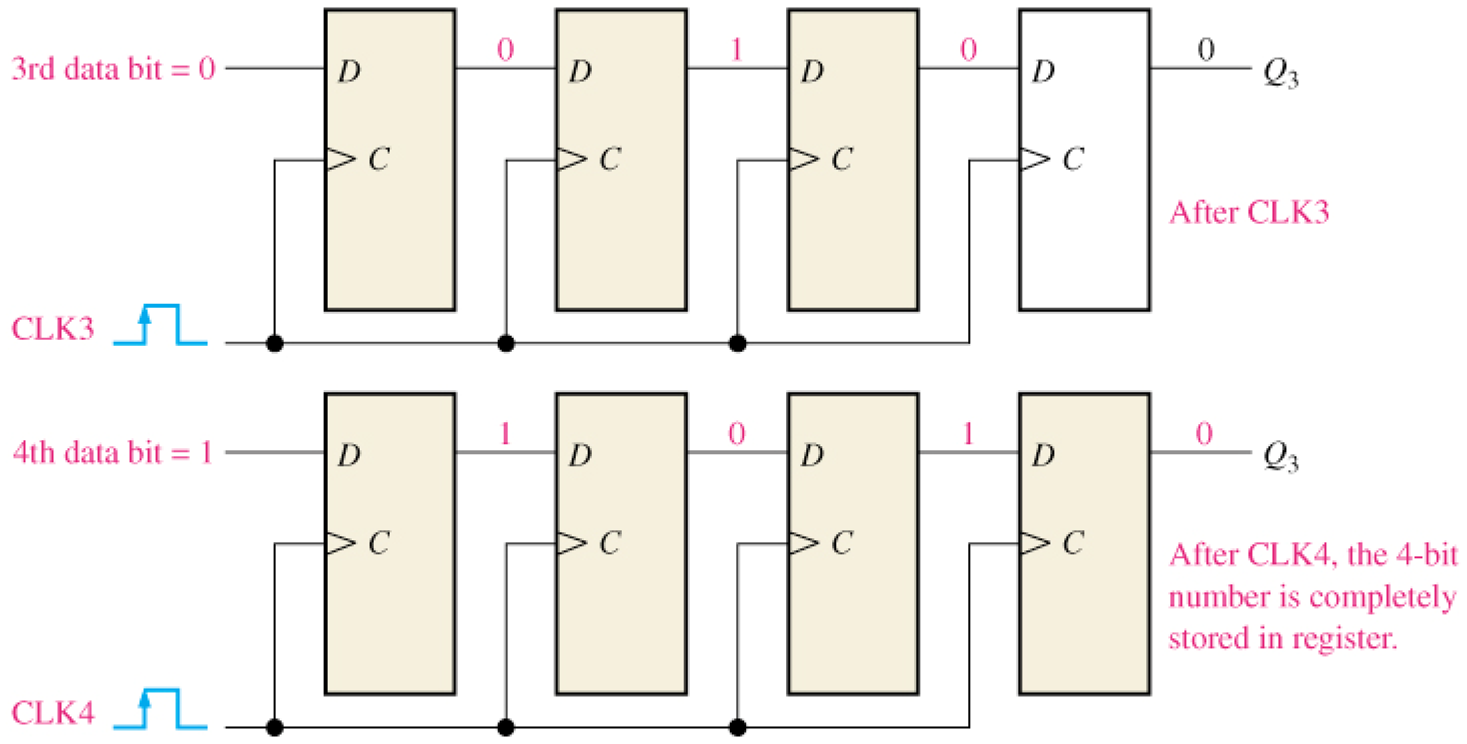
❑ Illustrative Example:

- Four bits(1010) being entered serially into the register



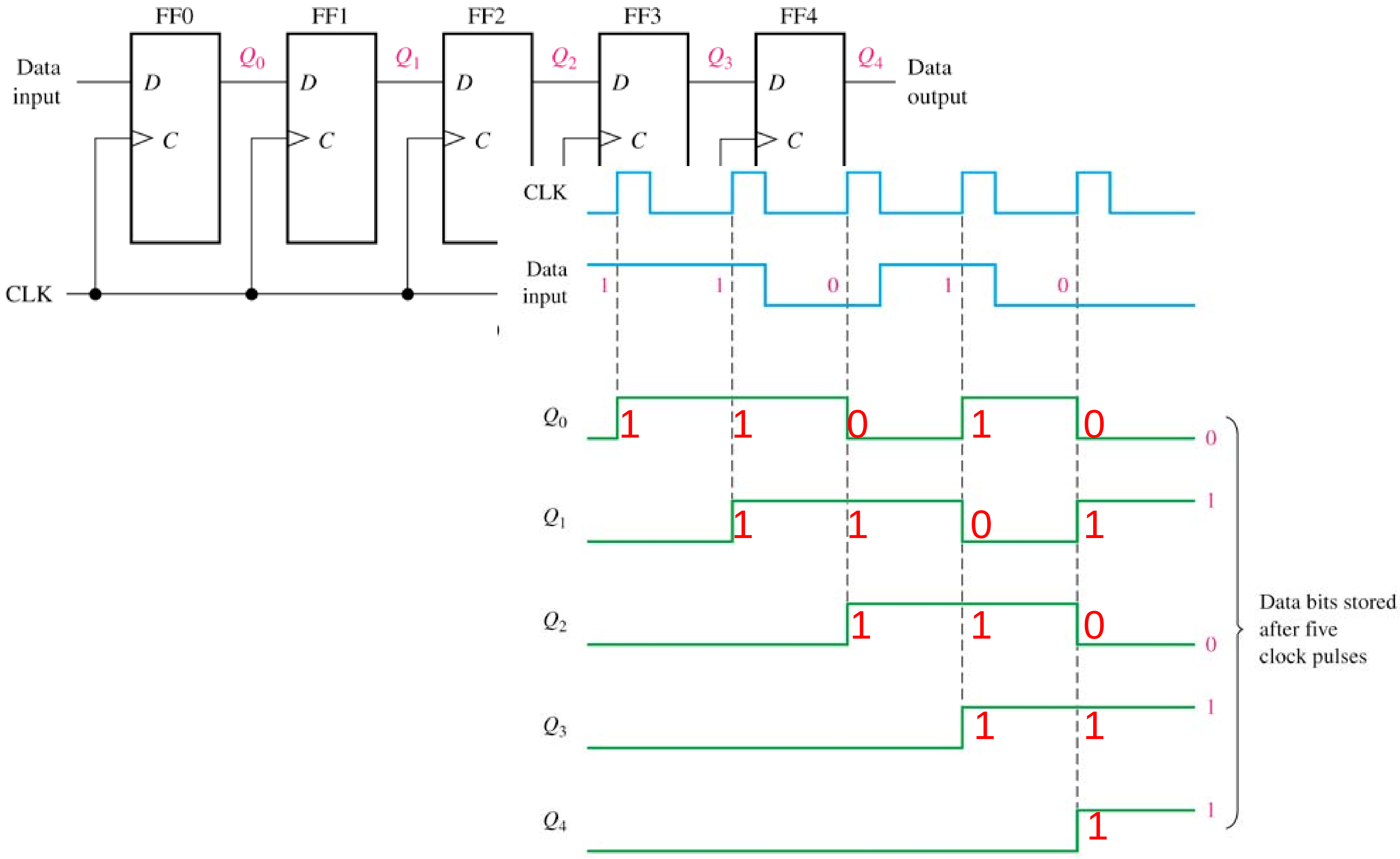
Serial In/Serial Out Shift Register

- Illustrative Example:
  - Four bits(1010) being entered serially into the register



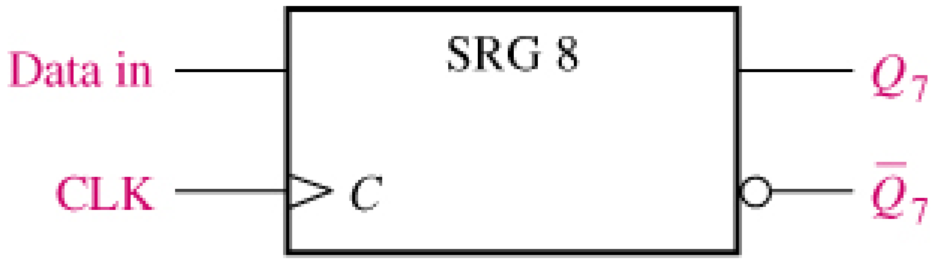
Serial In/Serial Out Shift Register

- Illustrative Example: Draw a Waveform



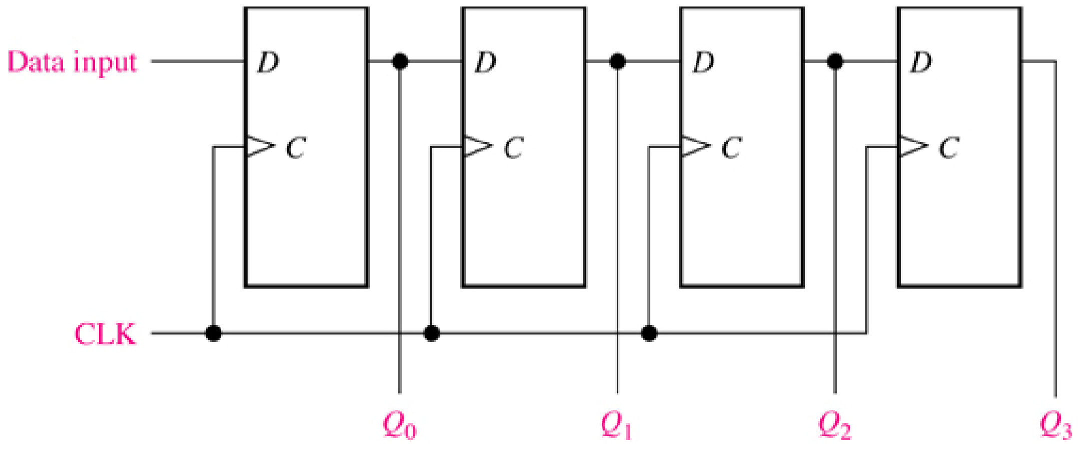
Serial In/Serial Out Shift Register

- Logic Symbol for an 8-bit Serial In/Serial Out Shift Register
  - SRG: Shift ReGister

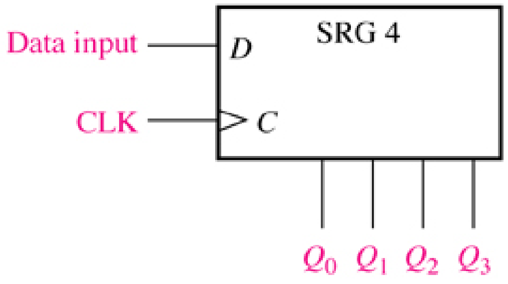


Serial In/Parallel Out Shift Register

- Operations
  - Data bits are entered serially (right-most bit first) into the register
  - Each data bit appears on its respective output line (all bits are available simultaneously)



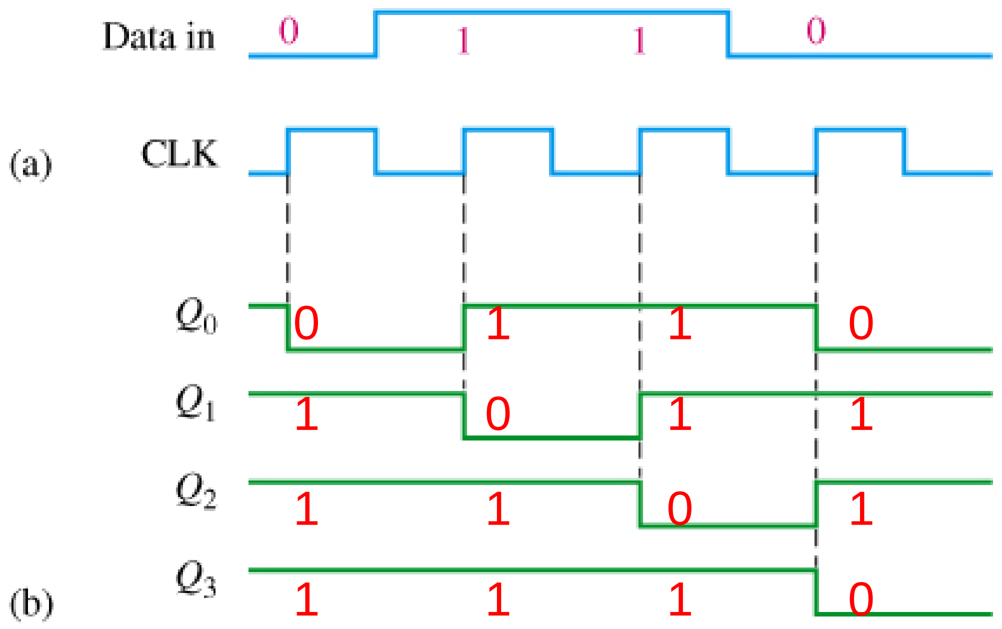
(a)

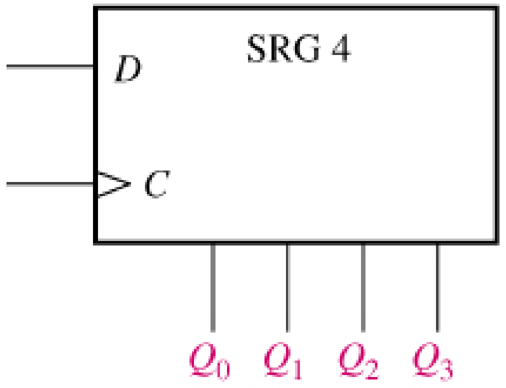


(b)

Serial In/Parallel Out Shift Register

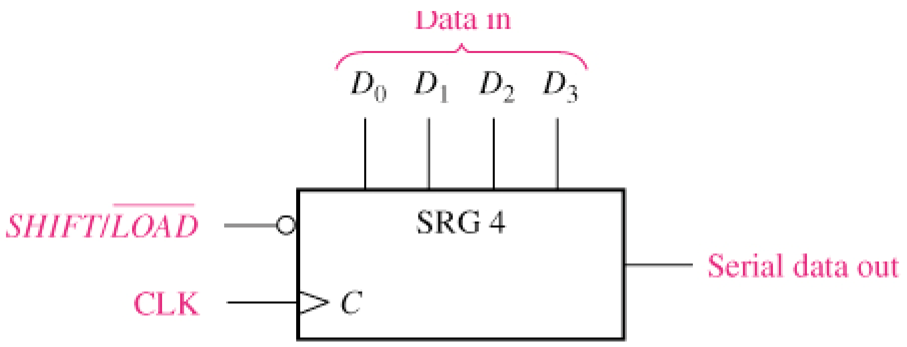
- Example: Draw a Waveform





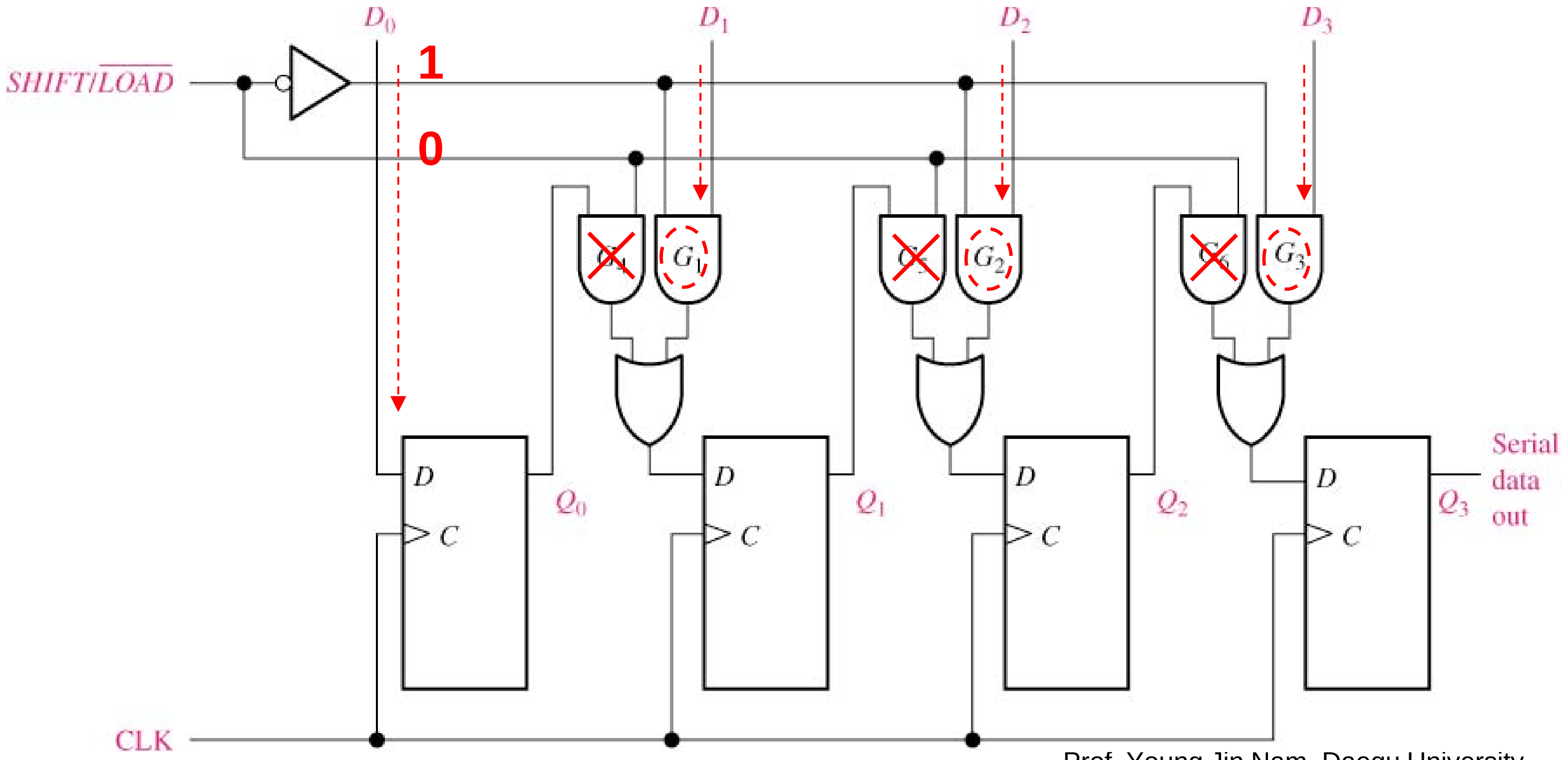
Parallel In/Serial Out Shift Register

- Operations
  - The data bits are entered simultaneously into their respective stages on parallel lines
  - One bit of data appears on an output line at a time
  - Four input lines (D0~D3), a SHIFT/LOAD’ input



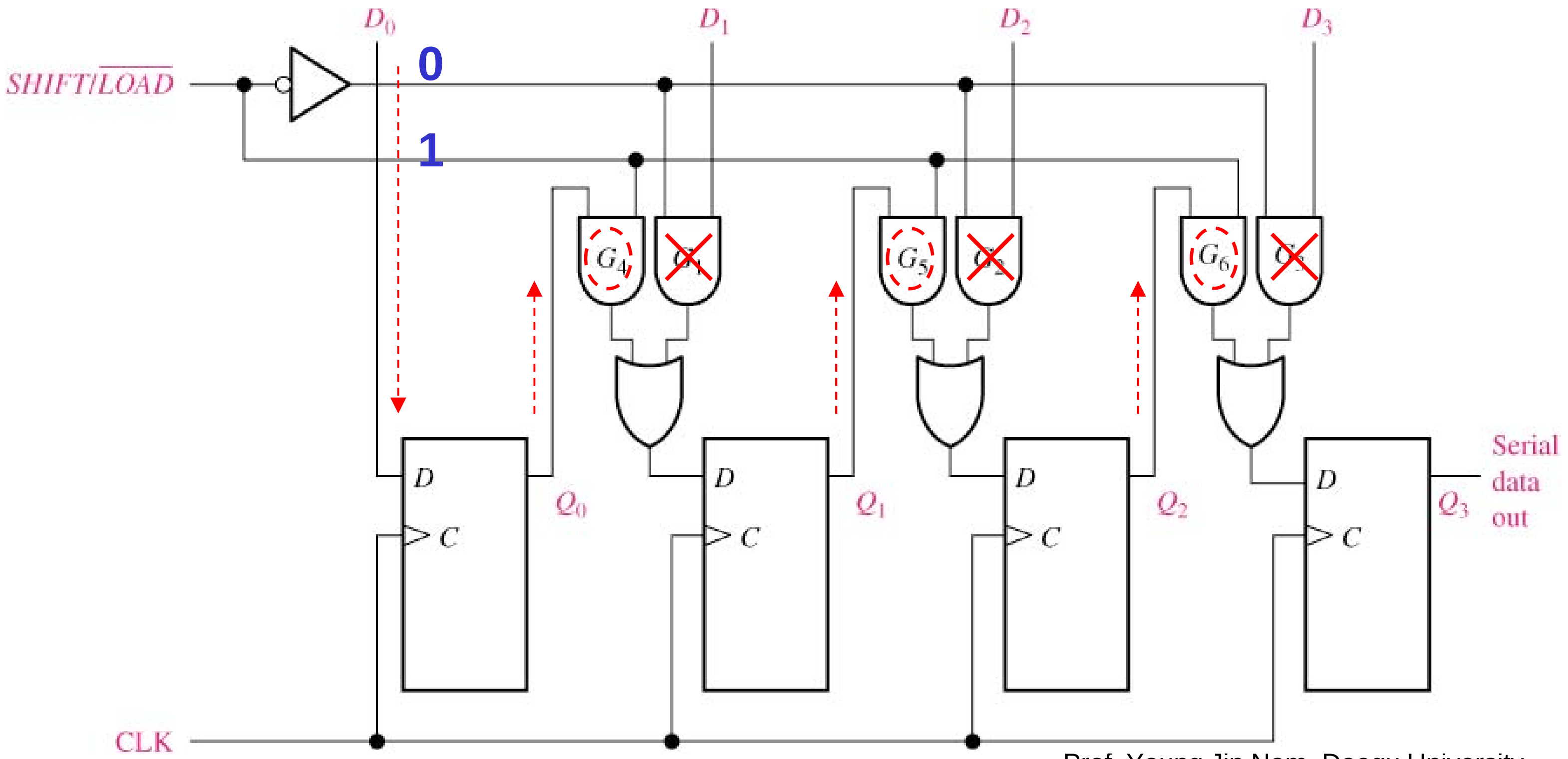
Parallel In/Serial Out Shift Register

- When SHIFT/LOAD’ = 0
  - Allow four bits of data to load in parallel into the register
  - Gates G1 through G3 are enabled
  - Allow each data bit to be applied to the D input of its respective F/F



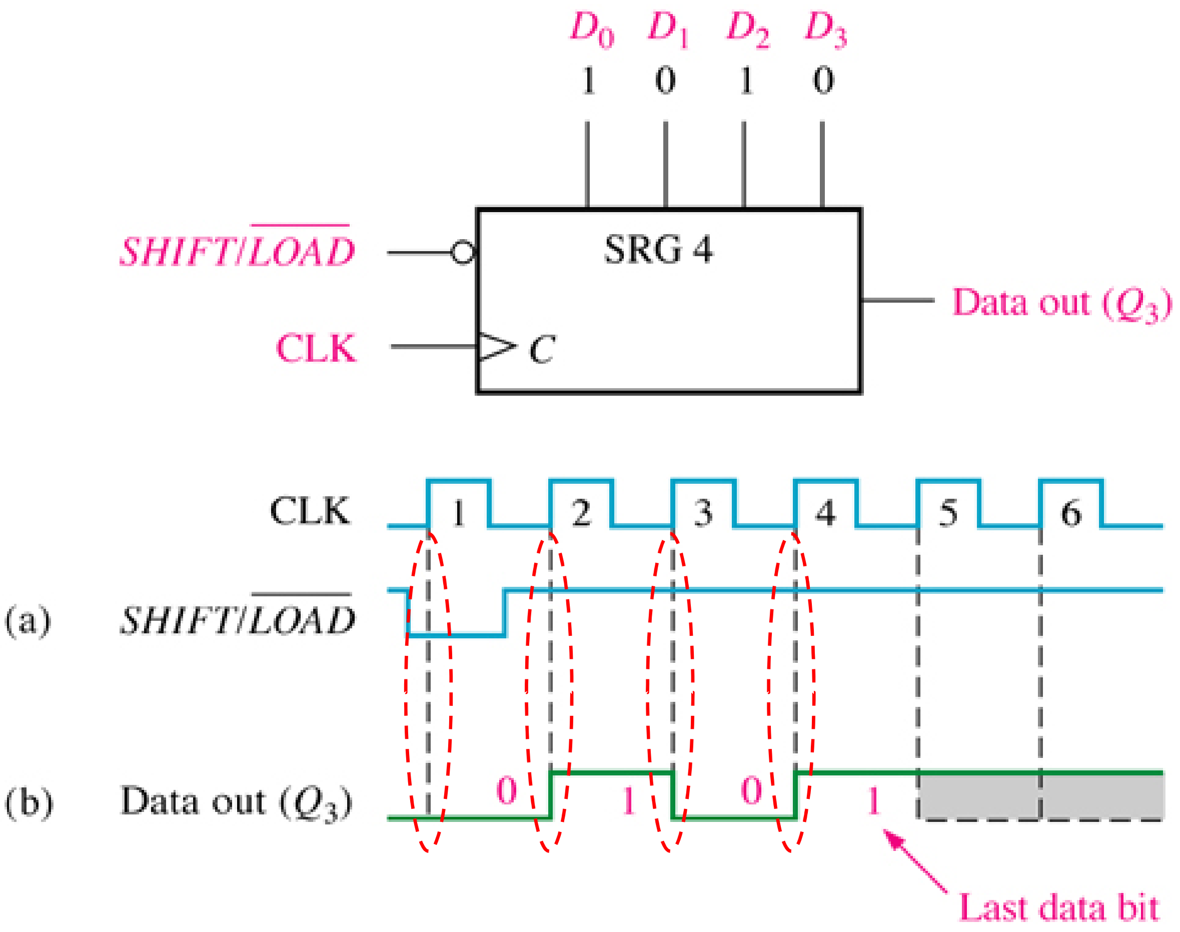
Parallel In/Serial Out Shift Register

- When SHIFT/LOAD' = 1
  - Allow the data bits to shift right from one stage to the next
  - Gates G4 through G6 are enabled



Parallel In/Serial Out Shift Register

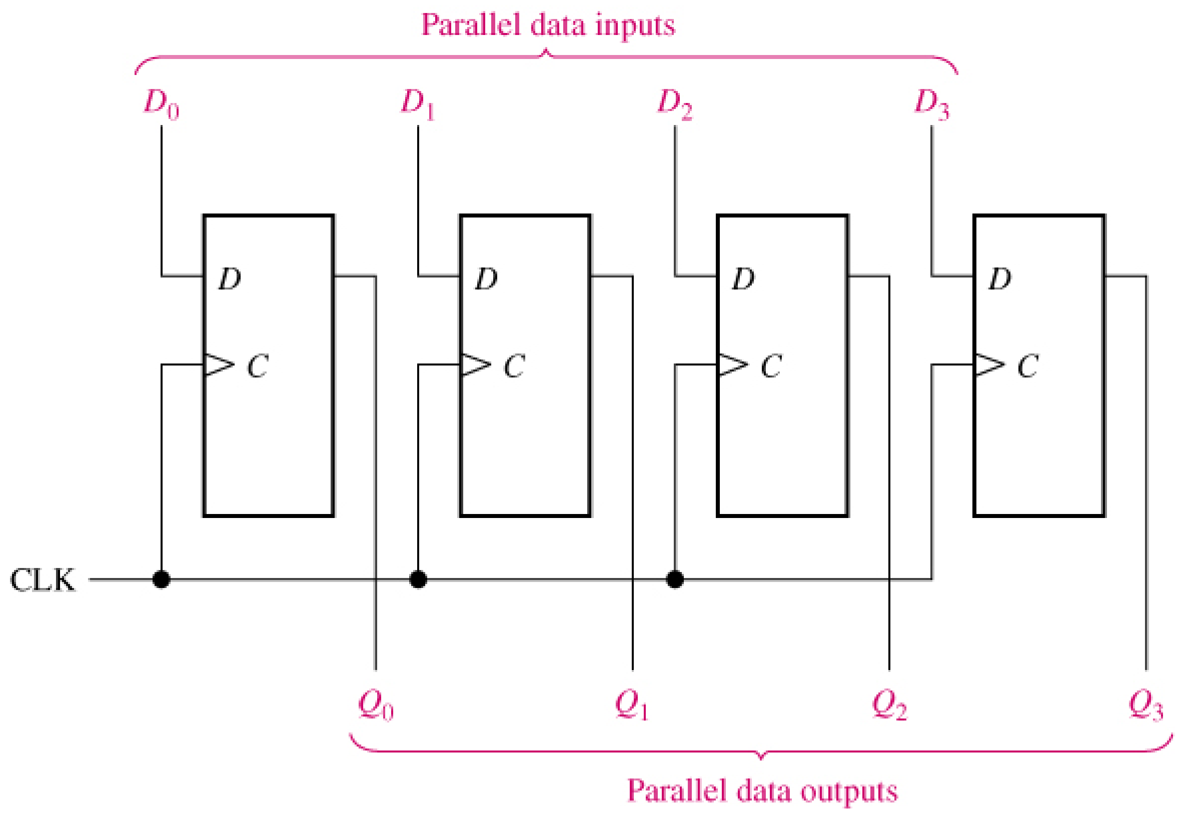
- Example: Draw a Waveform



Parallel In/ Parallel Out Shift Register

❑ Operations

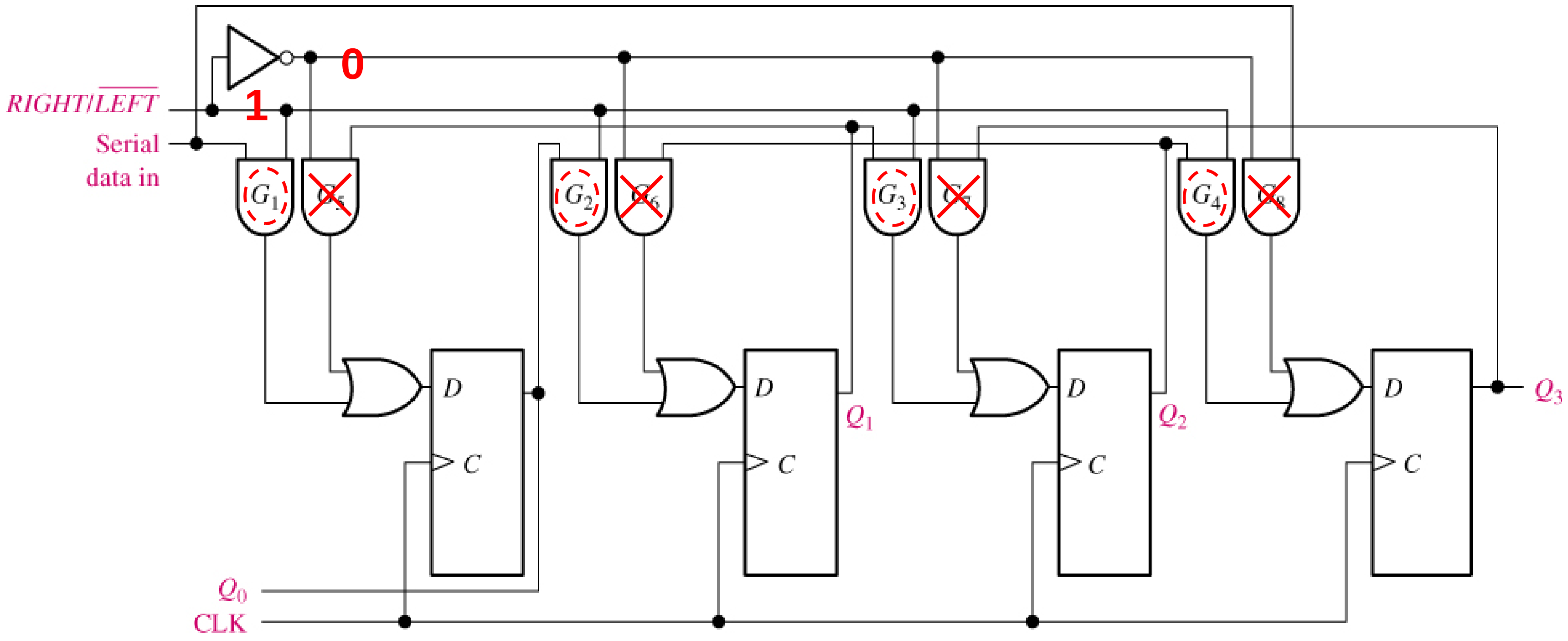
- Allow four bits of data to load in parallel into the register
- All bits are available simultaneously



Bidirectional Shift Registers

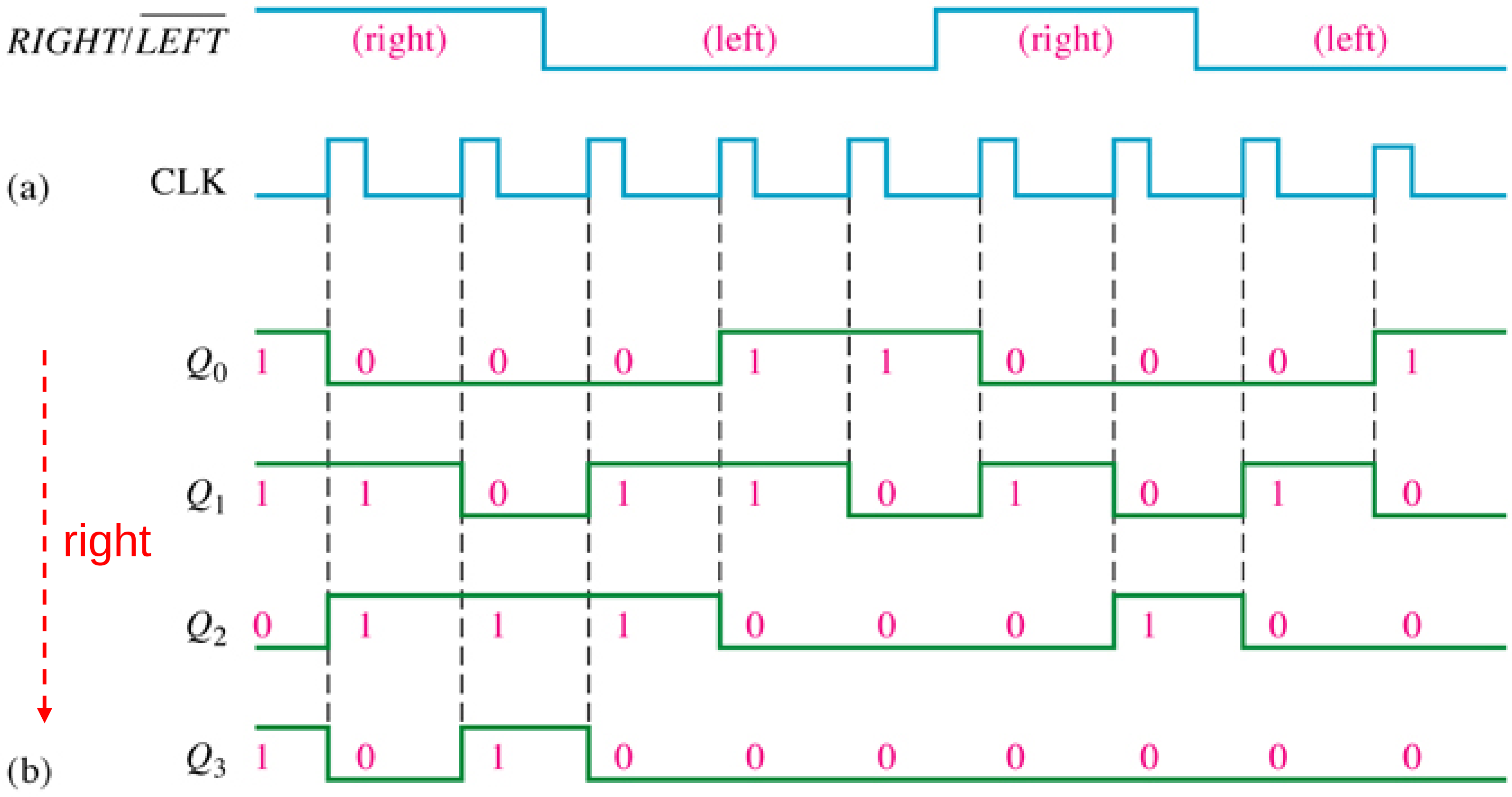
❑ Operations

- Data can shifted either left or right
- When RIGHT/LEFT’=1, data are to be shifted right
- When RIGHT/LEFT’=0, data are to be shifted left



Bidirectional Shift Registers

- Example: Draw a Waveform


RIGHT/$\overline{LEFT}$
(right)
(left)
(right)
(left)
(a)
CLK
Q_0 1 0 0 0 1 1 0 0 0 1
Q_1 1 1 0 1 1 0 1 0 1 0
right
Q_2 0 1 1 1 0 0 0 1 0 0
(b)
Q_3 1 0 1 0 0 0 0 0 0 0

Shift Register Counters

A Shift Register Counter

- A shift register with the serial output connected back to the serial input to produce special sequences
- Classified as counters because they exhibit a specified sequence of states
- Example: Johnson counter & ring counter

The Johnson Counter

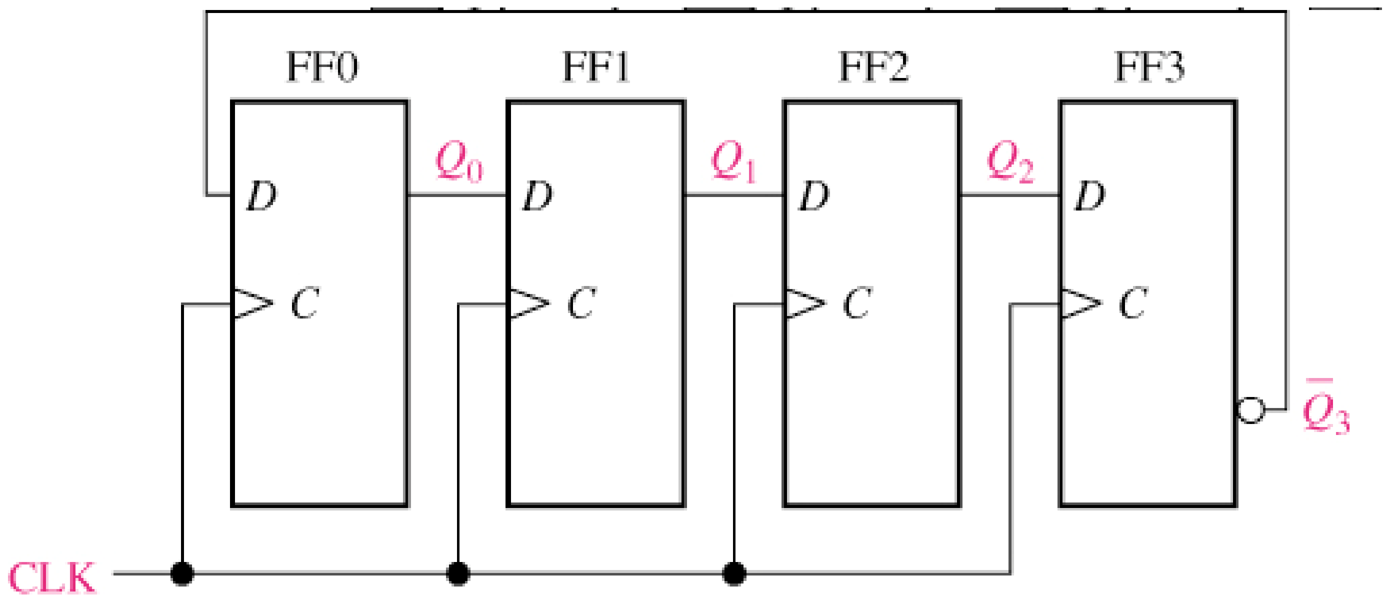
- Johnson Counter
  - The complement of the output of the last F/F is connected back to the D input of the first F/F
  - It produces a modulus of 2n, where n is the number of stages in the counter

- Example: Truth Table of 4-bit Johnson Sequence

<tab 10-1>

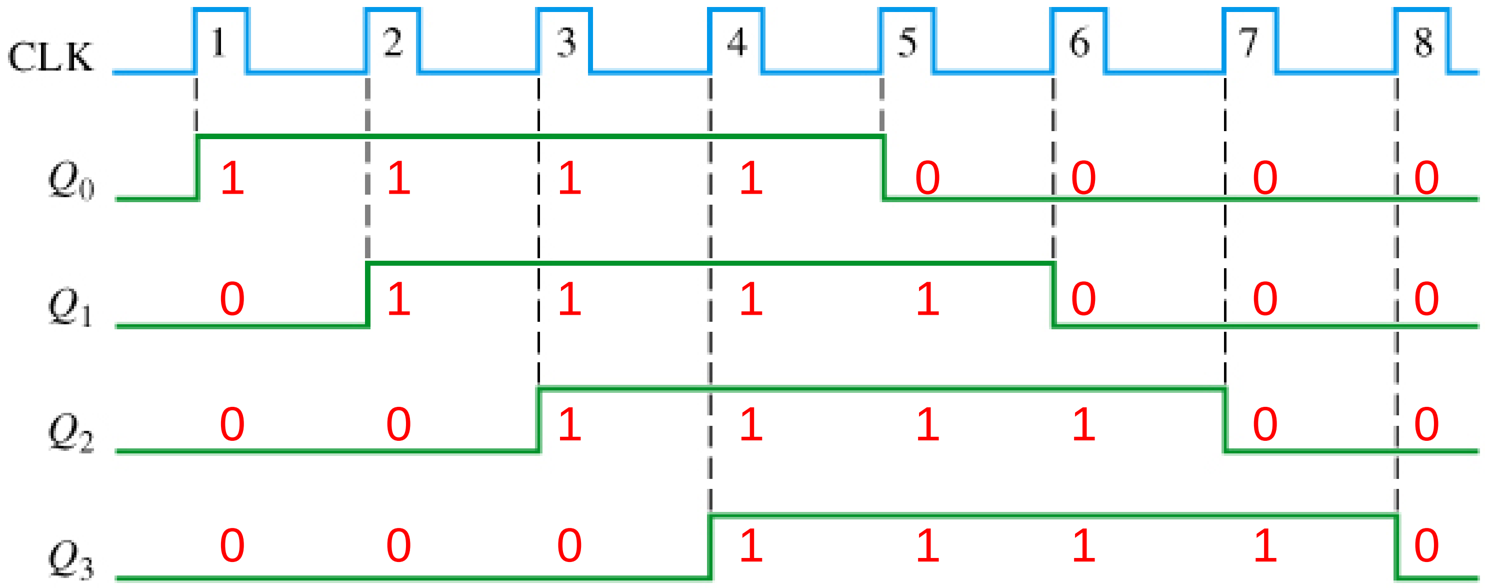
The Johnson Counter

- Block Diagram of 4-bit Johnson Counter



The Johnson Counter

- Timing Sequence of 4-bit Johnson Counter



The Johnson Counter

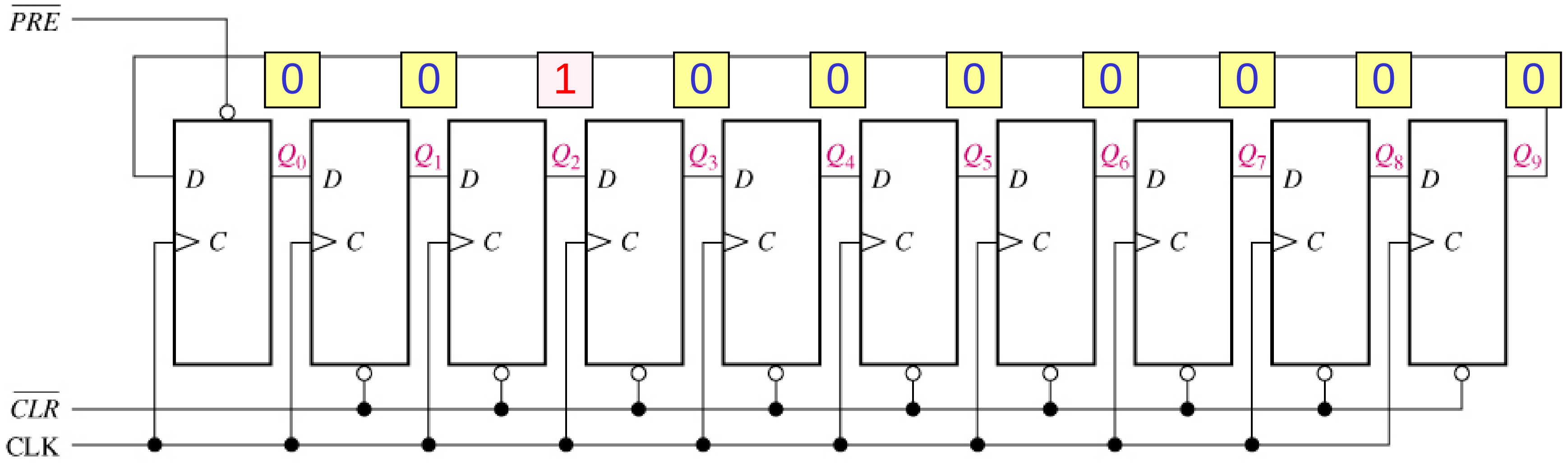
- Example: Truth Table of 5-bit Johnson Sequence

  <tab 10-2>

The Ring Counter

❑ Ring Counter

- Utilize one F/F for each state in its sequence
- Decoding gate is not required (a unique output for each decimal digit)
- Initially, 1 is preset into the first F/F & the rest are cleared



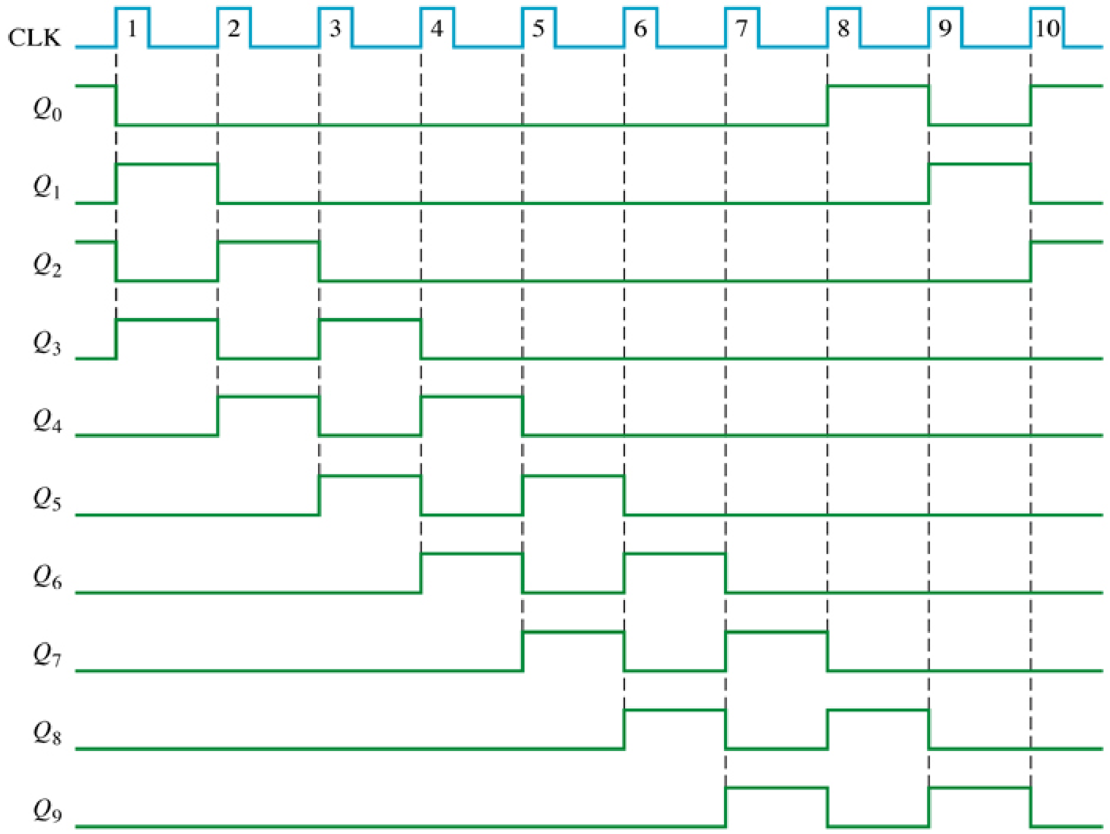
The Ring Counter

- Truth Table of 10-bit Ring Counter

  <tab 10-3>

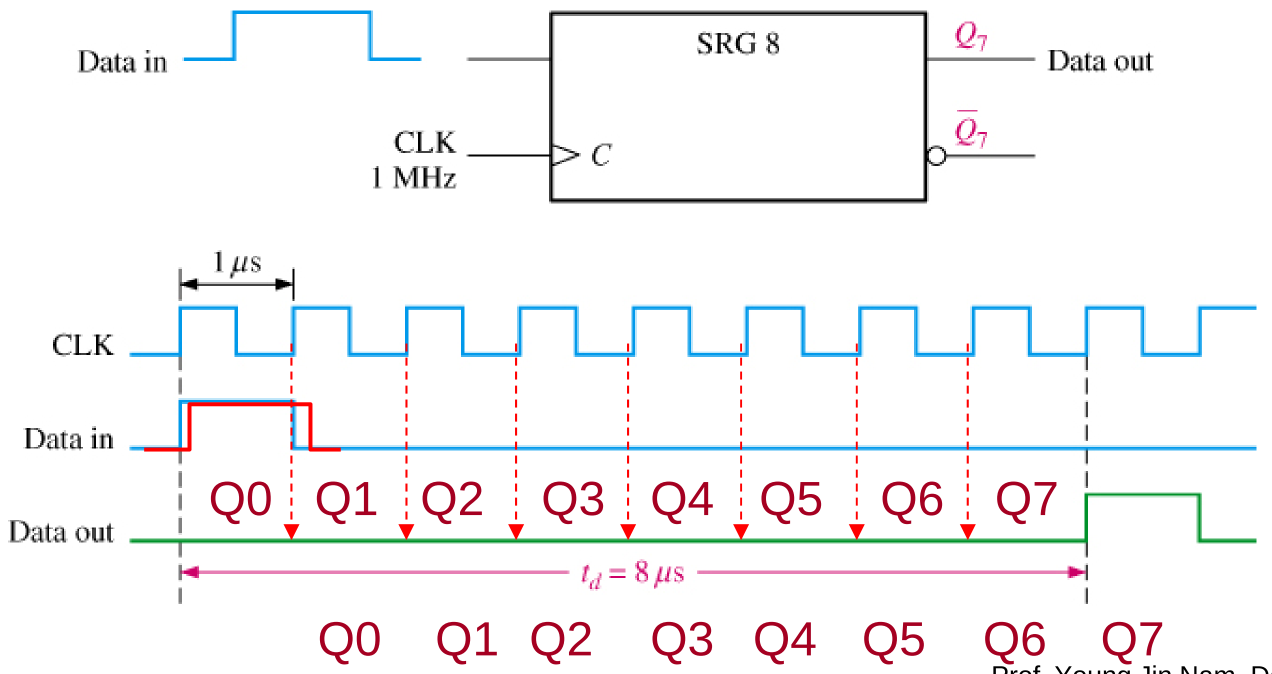
The Ring Counter

- Example: Draw a Waveform
  - The initial state = 1010000000



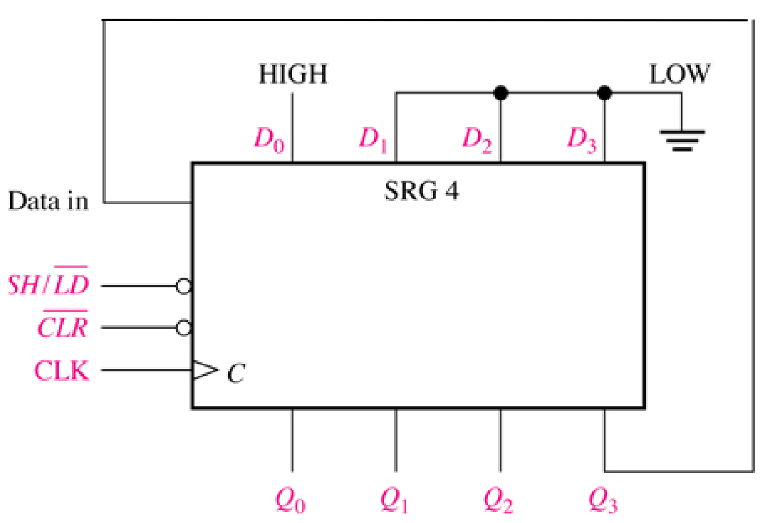
Application: Time Delay

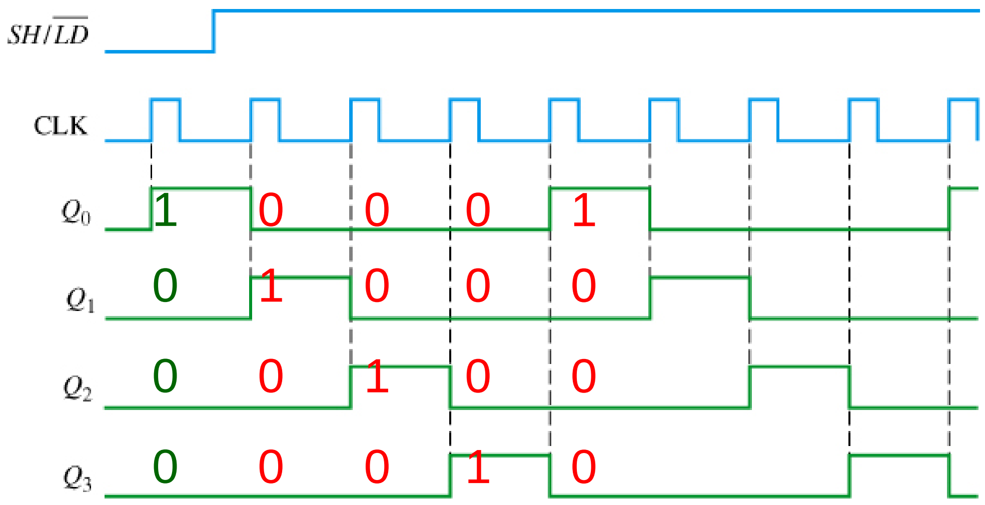
- Shift Register as a Time-Delay Device
  - Serial in/serial output shift register can be used to provide a time delay from input to output that is a function of both the # of stages(n) in the register & the clock frequency



Application: Ring Counter

- A Ring Counter using a Shift Register
  - If the output is connected back to the serial input, a shift register can be used as a ring counter
  - Initially, a bit pattern of 1000 can be synchronously preset into the counter (LD' = 0)

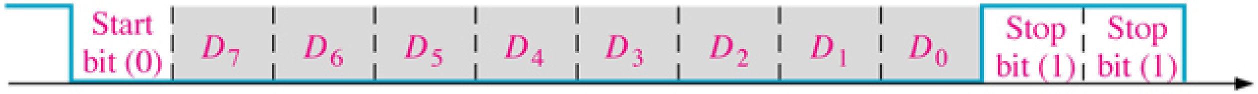




Application: Serial-to-Parallel Data Converter

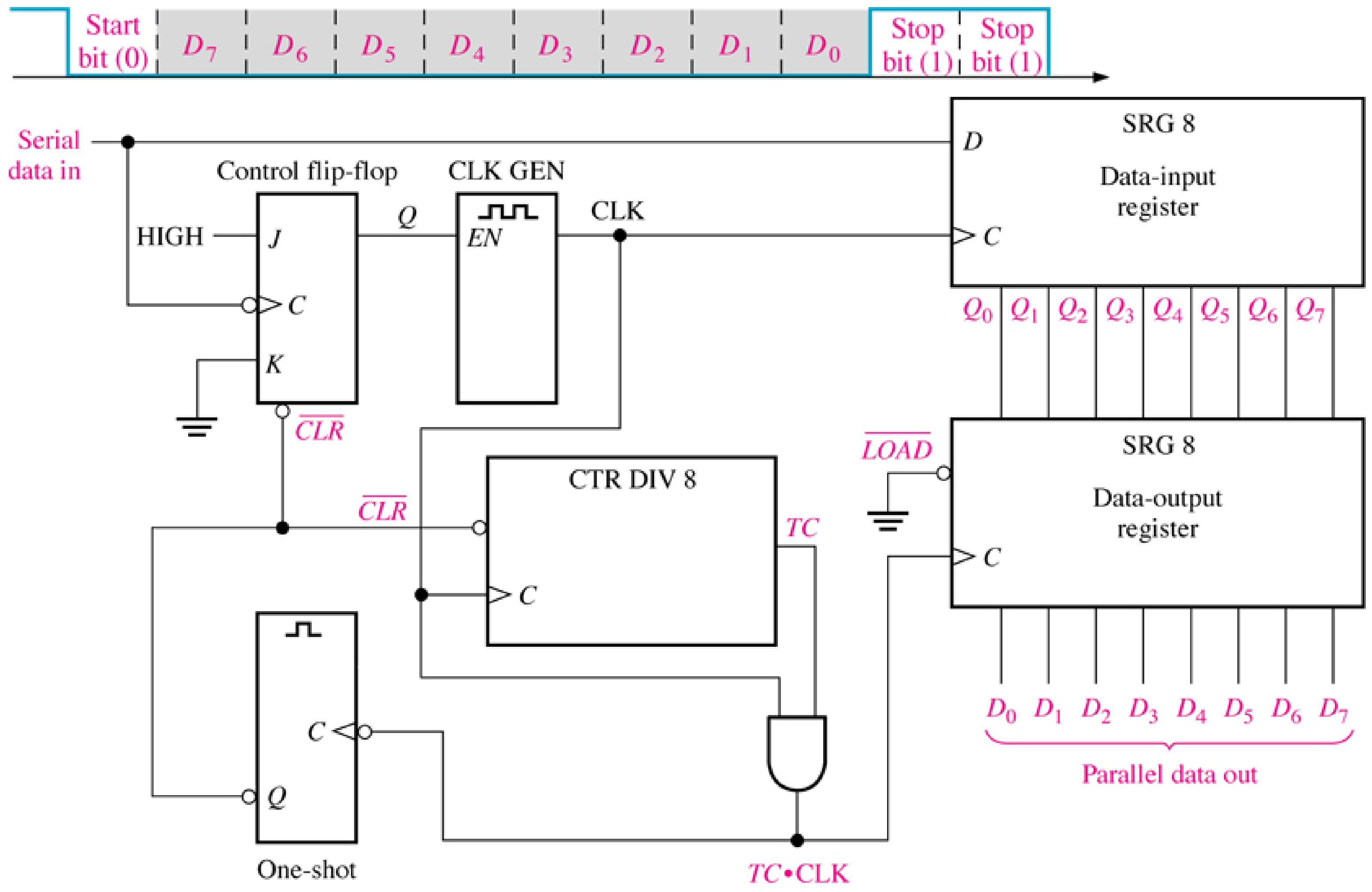
Simplified Serial-to-Parallel Data Converter

- Consists of 11 bits
- First bit(start bit) = 0 (beginning with a HIGH-to-LOW transition)
- Next 8 bits (D7~D0) are the data bits
- Last two bits (stop bits) are always 1s



Application: Serial-to-Parallel Data Converter

- Logic Diagram



The end of “**Shift Registers**”

Synchronous Sequential Circuit Design

Motivation

- Analysis of a few simple circuits

- Generalizes to Synchronous Sequential Circuits (SSC)
  - Outputs are Function of State (and Inputs)
  - Next States are Functions of State and Inputs
  - Used to implement circuits that control other circuits
  - "Decision Making" logic

- Application of Sequential Logic Design Techniques
  - Word Problems
  - Mapping into formal representations of SSC behavior
  - Case Studies

Overview

- Concept of the Synchronous Sequential Circuits
  - Partitioning into Datapath and Control
  - When Inputs are Sampled and Outputs Asserted
- Basic Design Approach
  - Six Step Design Process
- Alternative SSC Representations
  - State Diagram, VHDL
- Moore and Mealy Machines
  - Definitions, Implementation Examples
- Word Problems
  - Case Studies

Concept of the Synchronous Sequential Circuit

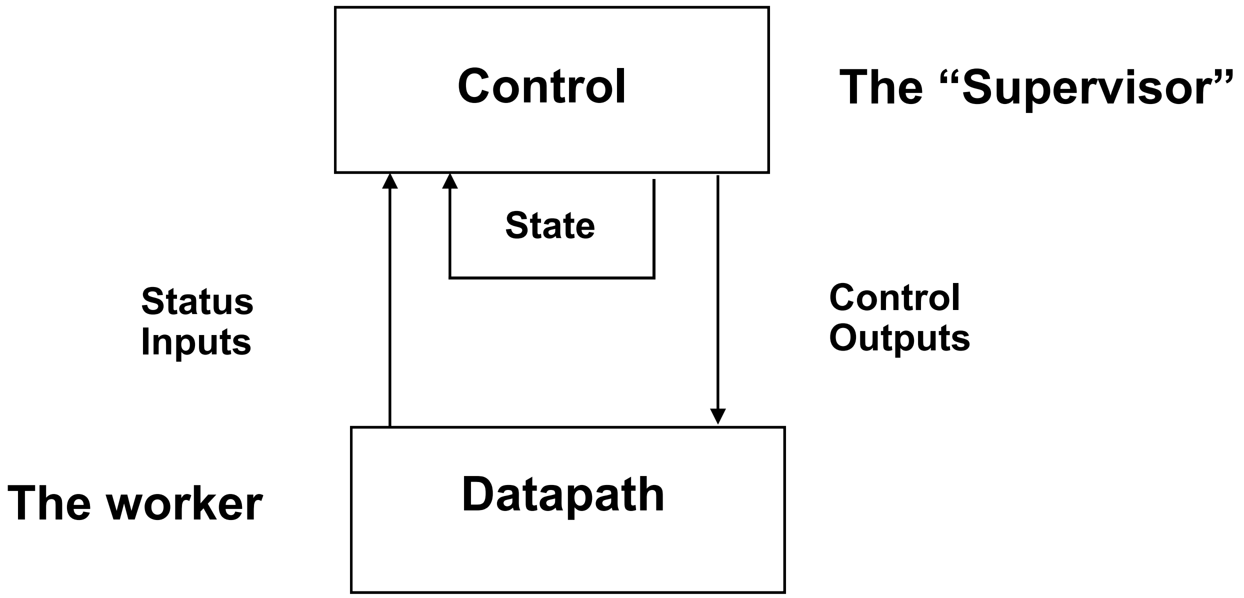
Complex Digital System = Datapath + Control

Registers
Combinational Functional Units (e.g., ALU)
Busses

Status →

← ***Control***

SSC generating sequences of control signals
Instructs datapath what to do next

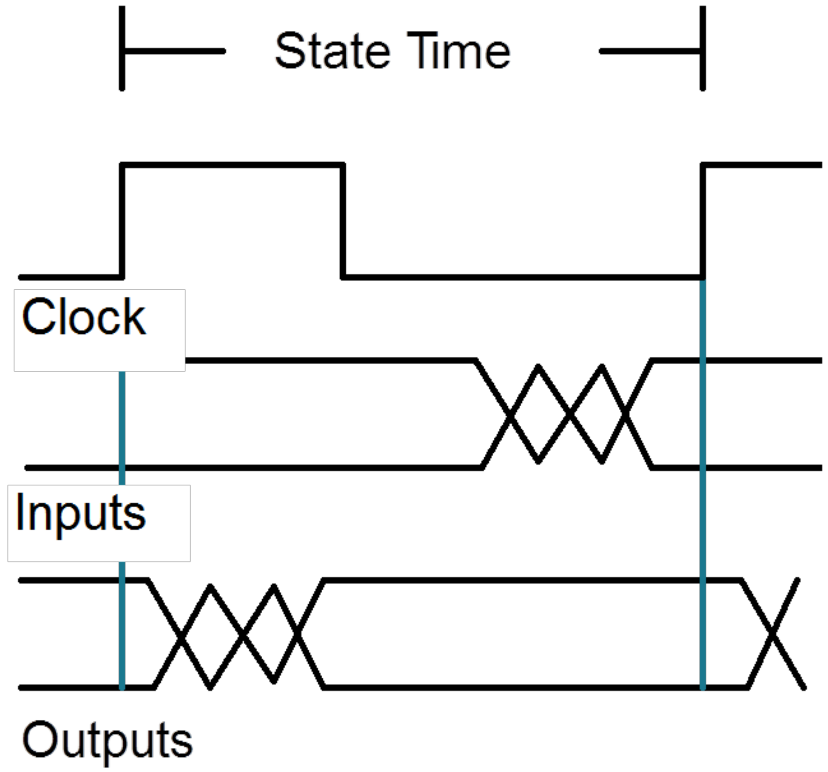


Concept of the Synchronous Sequential Circuit

- Timing: When are inputs sampled, next state computed, outputs asserted?
- State Time: Time between clocking events
- Clocking event causes state/outputs to transition, based on inputs
- For set-up/hold time considerations:
  - Inputs should be stable before clocking event
- After propagation delay, Next State entered, Outputs are stable
  - NOTE: Asynchronous output (Mealy) take effect immediately
  - Synchronous outputs (Moore) take effect at the next clocking event
    - E.g., tri-state enable: effective immediately
    - sync. counter clear: effective at next clock event

Concept of the Synchronous Sequential Circuit

Example: **Positive Edge Triggered Synchronous System**



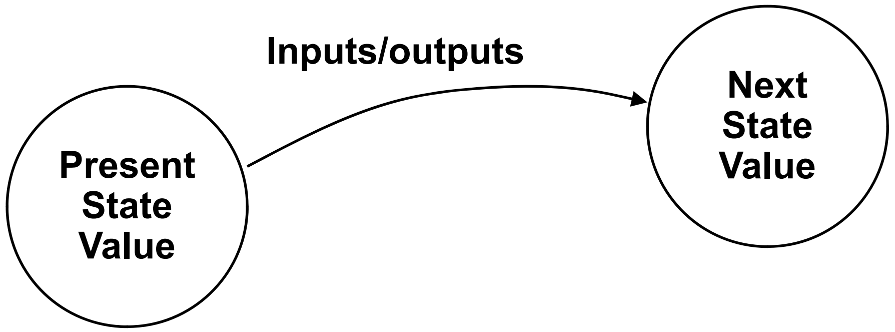
- **On rising edge, inputs sampled; outputs, next state computed**
- **After propagation delay, outputs and next state are stable**
- ***Immediate Outputs:***
  - **affect datapath immediately**
  - **could cause inputs from datapath to change**
- ***Delayed Outputs:***
  - **take effect on next clock edge**
  - **propagation delays must exceed hold times**

Sequential Circuit Analysis

- Start with schematic diagram
- Need to determine how circuit works
  - Trace schematic, determine equations of operation
    - FF input equations
    - sequential circuit output equations
  - Create State transition table
    - Sequential circuit inputs, FFs are comb. logic inputs
    - Organize truth table as current state (FFs) and inputs
    - Create FF input, seq. Circuit output columns
    - From FF char. Tables, determine FF next state values

Sequential Circuit Analysis (cont.)

- Generate State Diagram
  - Circles (nodes) represent current or present state values
  - Lines (arcs) represent how state and output values change
    - Given the current state and current inputs, the next state and output values are indicated by the associated arc
  - State diagram can have different forms depending on the type of sequential circuit output.



Basic Design Approach

- ***Six Step Process***

  1. **Understand the statement of the Specification**
  2. **Obtain an abstract specification of the SSC**
  3. **Generate State Table**
  4. **Perform state assignment**
  5. **Choose FF types to implement SSC state register**
  6. **Implement the SSC**

Basic Design Approach

Example: **Vending Machine SSC**

General Machine Concept:

deliver package of gum after 15 cents deposited

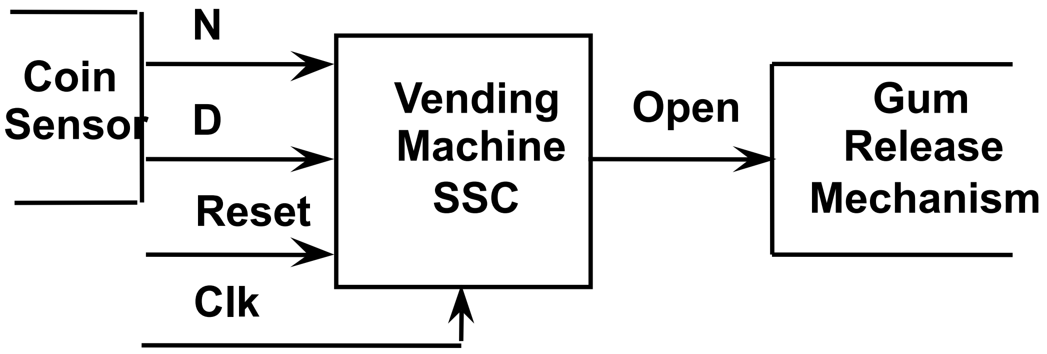
single coin slot for dimes, nickels

no change

Step 1. ***Understand the problem:***

Draw a picture!

Block Diagram



Vending Machine Example

Step 2. ***Map into more suitable abstract representation***

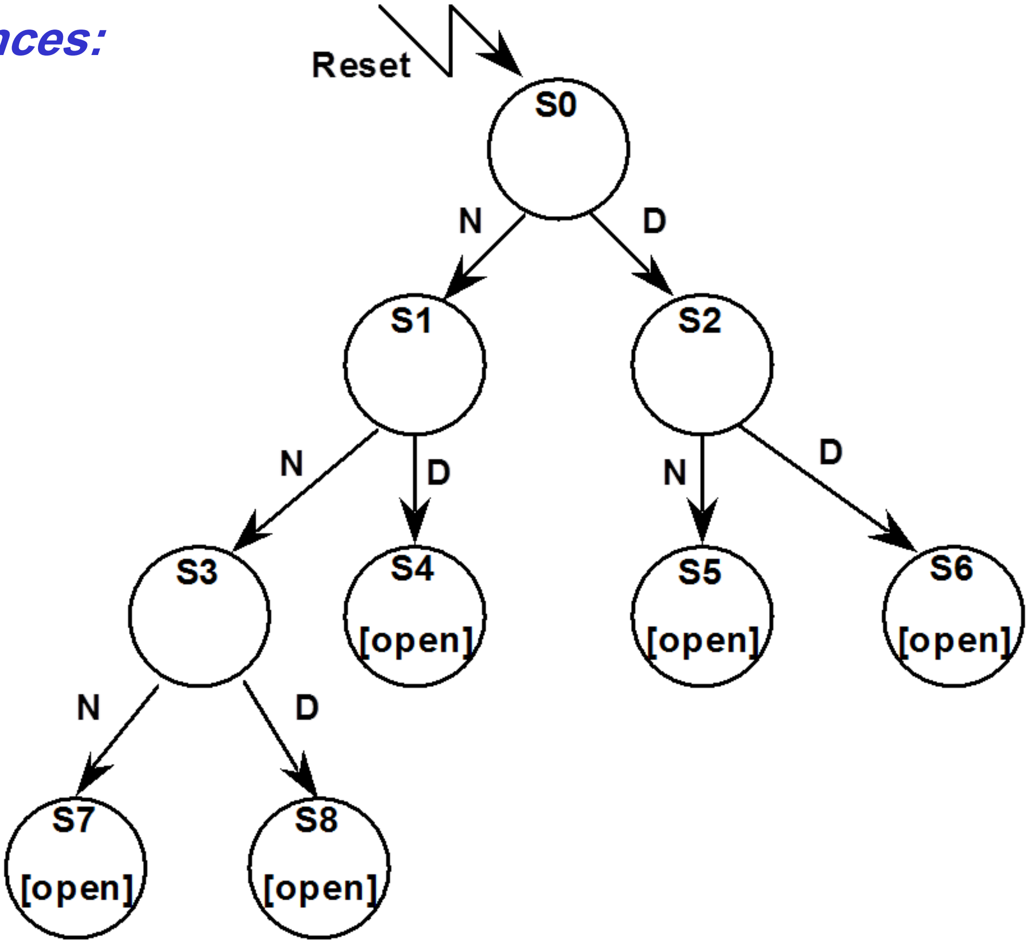
Tabulate typical input sequences:

three nickels
nickel, dime
dime, nickel
two dimes
two nickels, dime

Draw state diagram:

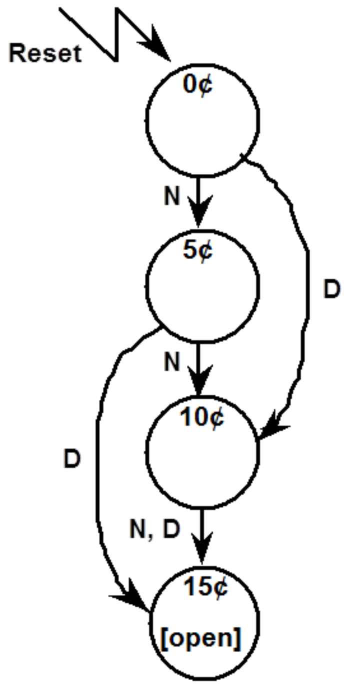
Inputs: N, D, reset

Output: open



Vending Machine Example

Step 3: State Minimization



reuse states whenever possible

Present State	Inputs D	Inputs N	Next State	Output Open
0¢	0	0	0¢	0
	0	1	5¢	0
	1	0	10¢	0
	1	1	X	X
5¢	0	0	5¢	0
	0	1	10¢	0
	1	0	15¢	0
	1	1	X	X
10¢	0	0	10¢	0
	0	1	15¢	0
	1	0	15¢	0
	1	1	X	X
15¢	X	X	15¢	1

Symbolic State Table

Vending Machine Example

Step 4: State Encoding

State	Present State Q_1	Present State Q_0	Inputs D	Inputs N	Next State Q_1^+ (D_1)	Next State Q_0^+ (D_0)	Output Open
0¢	0	0	0	0	0	0	0
			0	1	0	1	0
			1	0	1	0	0
			1	1	X	X	X
5¢	0	1	0	0	0	1	0
			0	1	1	0	0
			1	0	1	1	0
			1	1	X	X	X
10¢	1	0	0	0	1	0	0
			0	1	1	1	0
			1	0	1	1	0
			1	1	X	X	X
15¢	1	1	0	0	1	1	1
			0	1	1	1	1
			1	0	1	1	1
			1	1	X	X	X

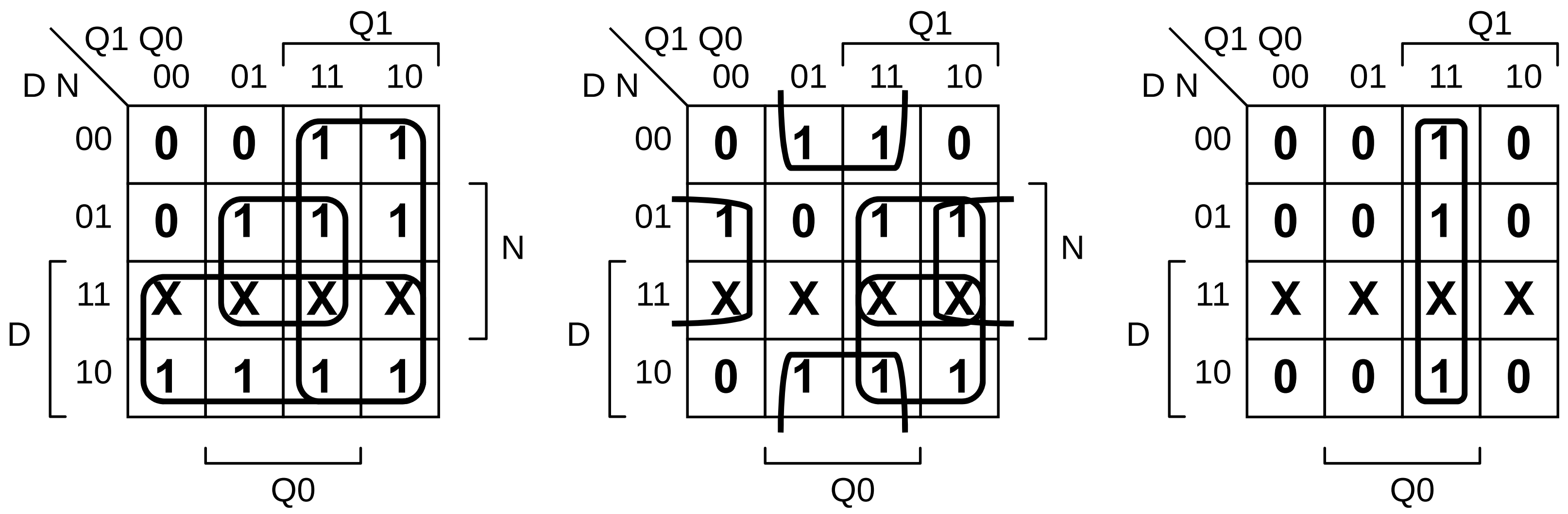
NOTE!

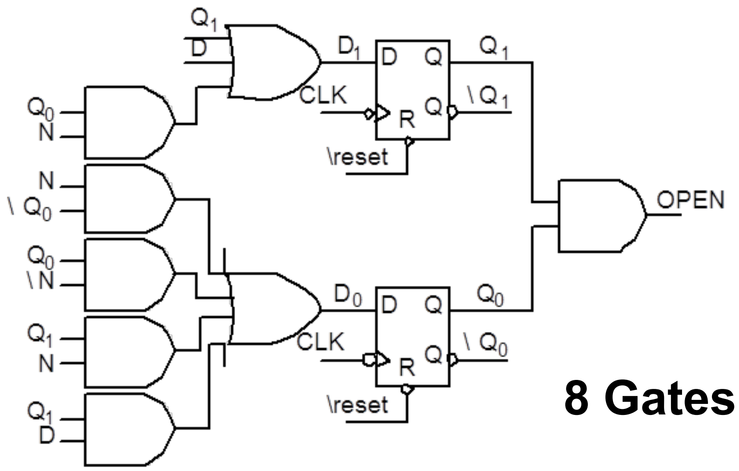
For D-FFs the next state will be what is at the D input. So each FF's next state values in the state table must be the D inputs for that FF.

Vending Machine Example

Step 5. *Choose FFs for implementation*

D FF easiest to use





8 Gates

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

$$D0 = N\,\overline{Q0} + Q0\,\overline{N} + Q1\,N + Q1\,D$$

$$OPEN = Q1\,Q0$$

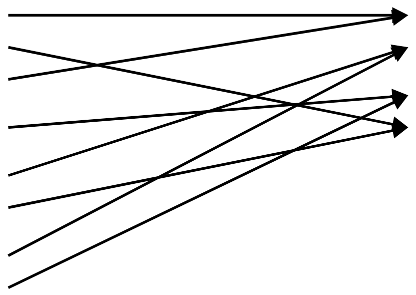
Designing with SR, JK, and T Flip-Flops

- Sequential design with D-FFs is easy; next state depends on D input only
- We can use other FFs but the process is a little more involved
  - State table defines set of present state to next state transitions
  - What we need to design the next state combinational logic is the FF input values needed for each Q $\rightarrow$ Q+ transition
- This table is known as the FF excitation table
  - Derived from the FF characteristic table

Derivation of JK Excitation Table

JK Characteristic Table

J	K	Q	Q+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0



JK Excitation Table

Q	Q+	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

Flip-Flop Excitation Tables

Q	Q+	J	K	S	R	T	D
0	0	0	X	0	X	0	0
0	1	1	X	1	0	1	1
1	0	X	1	0	1	1	0
1	1	X	0	X	0	0	1

You can use any FF type for your implementation

FF types can be mixed; I.e. in vending machinge you could use a JK FF for Q_1 and a T FF for Q_0

Vending Machine Example

Step 5. *Choosing FF for Implementation*

J-K FF

JK Excitation Table

Q	**Q+**	**J**	**K**
0	**0**	**0**	**X**
0	**1**	**1**	**X**
1	**0**	**X**	**1**
1	**1**	**X**	**0**

Present State Q_1	Q_0	Inputs D	N	Next State Q_1^+	Q_0^+	J_1	K_1	J_0	K_0
0	0	0	0	0	0	0	X	0	X
		0	1	0	1	0	X	1	X
		1	0	1	0	1	X	0	X
		1	1	X	X	X	X	X	X
0	1	0	0	0	1	0	X	X	0
		0	1	1	0	1	X	X	1
		1	0	1	1	1	X	X	0
		1	1	X	X	X	X	X	X
1	0	0	0	1	0	X	0	0	X
		0	1	1	1	X	0	1	X
		1	0	1	1	X	0	1	X
		1	1	X	X	X	X	X	X
1	1	0	0	1	1	X	0	X	0
		0	1	1	1	X	0	X	0
		1	0	1	1	X	0	X	0
		1	1	X	X	X	X	X	X

Remapped encoded state transition table using JK excitation table

Vending Machine Example

Implementation:

DN \ Q1 Q0	00	01	11	10
00	0	0	X	X
01	0	1	X	X
11	X	X	X	X
10	1	1	X	X

DN \ Q1 Q0	00	01	11	10
00	X	X	0	0
01	X	X	0	0
11	X	X	X	X
10	X	X	0	0

DN \ Q1 Q0	00	01	11	10
00	0	X	X	0
01	1	X	X	1
11	X	X	X	X
10	0	X	X	1

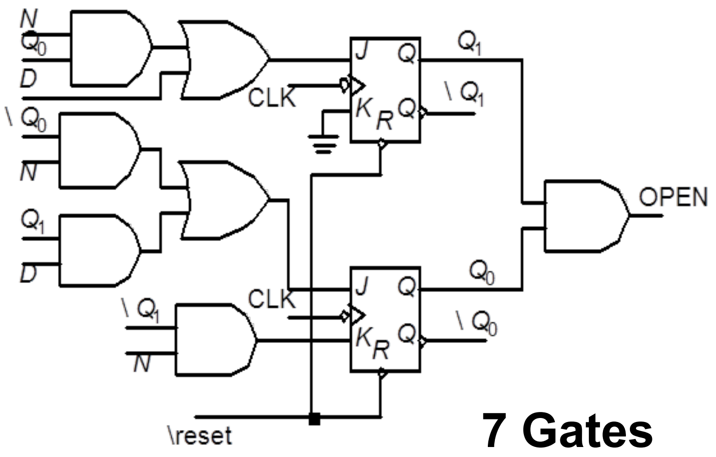
DN \ Q1 Q0	00	01	11	10
00	X	0	0	X
01	X	1	0	X
11	X	X	X	X
10	X	0	0	X

J1 = D + Q0 N

K1 = 0

J0 = N + Q1 D

K0 = $\overline{\text{Q1}}$ N



7 Gates

Moore vs. Mealy Machines

Definitions

Moore Machine

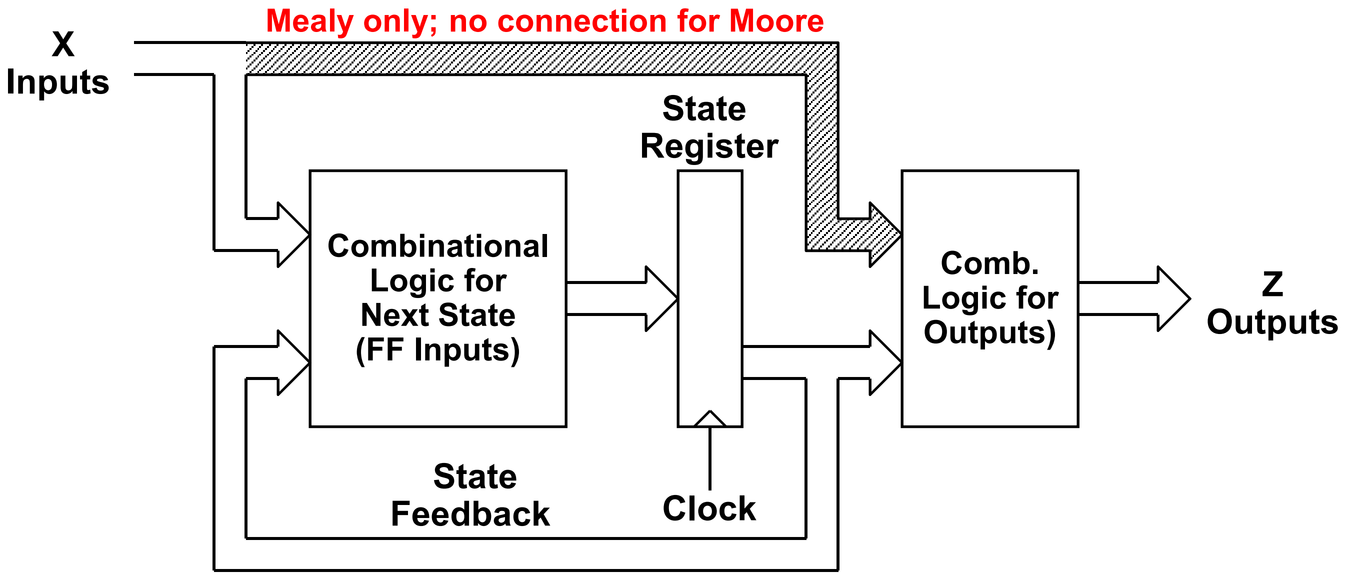
Outputs are function solely of the current state

Outputs change synchronously with state changes

Mealy Machine

Outputs depend on state AND inputs

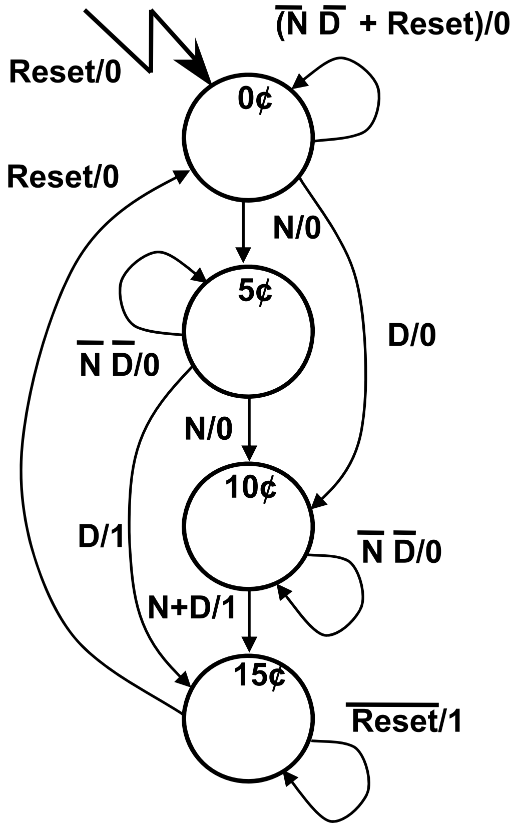
Input change causes an immediate (asynchronous) output change



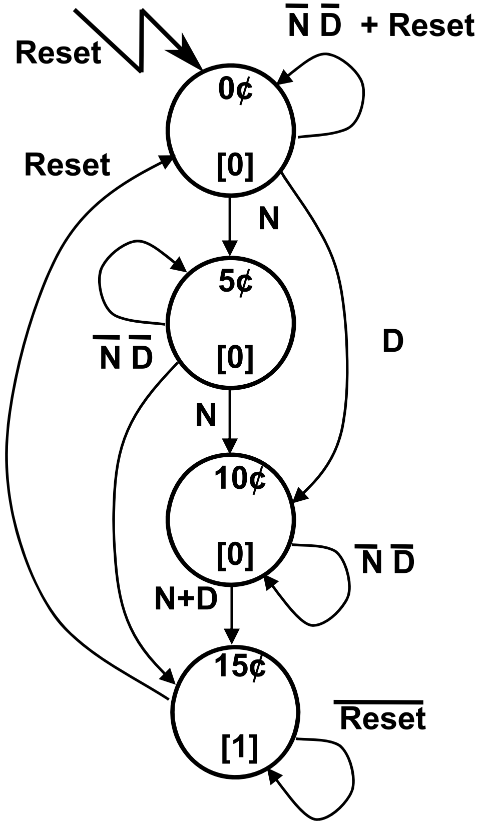
Moore and Mealy Machines

State Diagram Equivalents

Mealy Machine



Moore Machine



Outputs are associated with Transitions

Outputs are associated with State

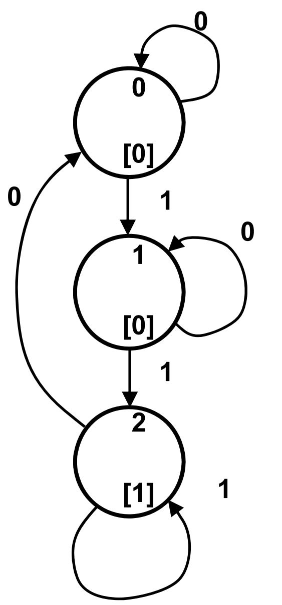
Moore and Mealy Machines

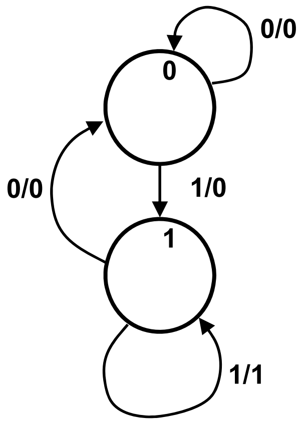
States vs. Transitions

Mealy Machine typically has fewer states than Moore Machine for same output sequence

Same I/O behavior

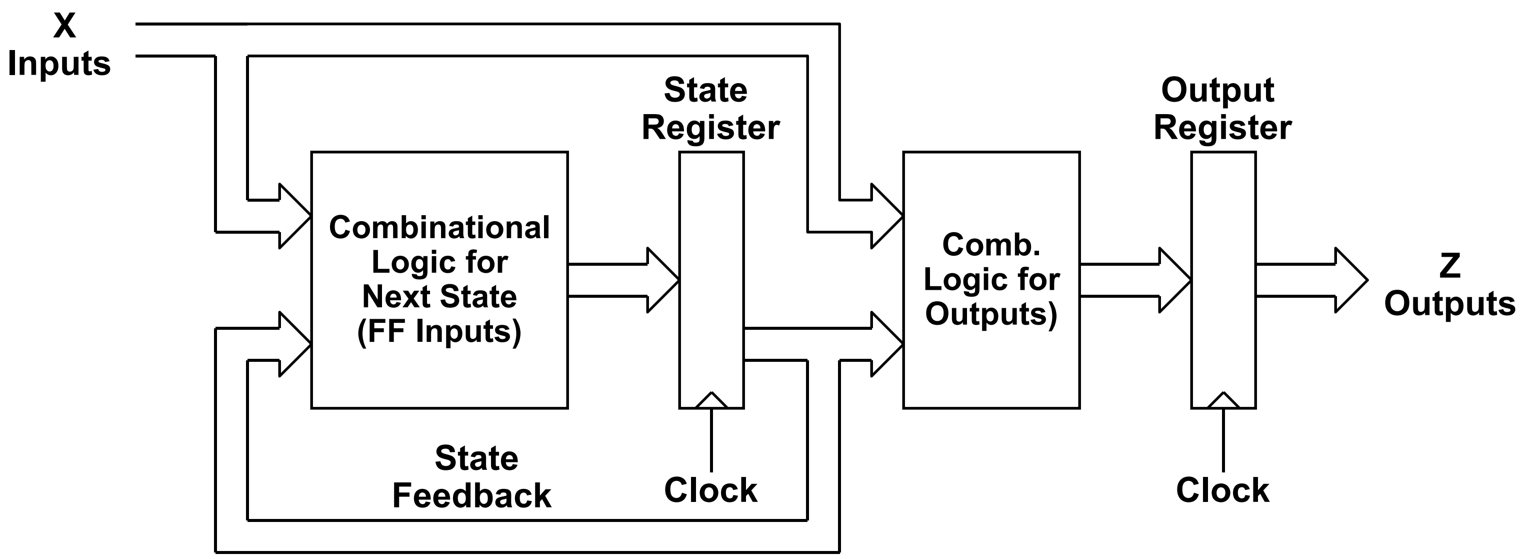
Different # of states





Moore and Mealy Machines

Synchronous Mealy Machine



Latched state AND outputs
Avoids glitchy outputs!
Outputs are delayed by up to 1 clock period
Usually equivalent to the Moore form

Synchronous Sequential Circuit Word Problems

Mapping English Language Description to Formal Specifications

Four Case Studies:

- **Finite String Pattern Recognizer**
- **Complex Counter with Decision Making**
- **Traffic Light Controller**
- **Digital Combination Lock**

Synchronous Sequential Circuit Word Problems

Finite String Pattern Recognizer

A finite string recognizer has one input (X) and one output (Z). The output is asserted whenever the input sequence …010… has been observed, as long as the sequence 100 has never been seen.

Step 1. Understanding the problem statement

Sample input/output behavior:

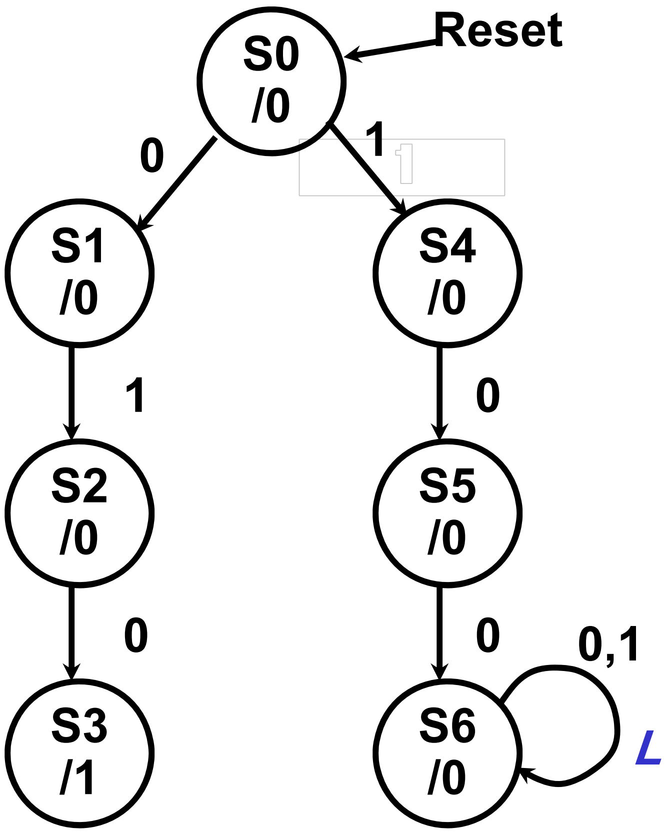
```
X:   00101010010…
Z:   00010101000…

X:   11011010010…
Z:   00000001000…
```

Synchronous Sequential Circuit Word Problems

Finite String Recognizer

Step 2. Draw State Diagrams for the strings that must be recognized. I.e., 010 and 100.



Moore State Diagram
Reset signal places SSC in S0

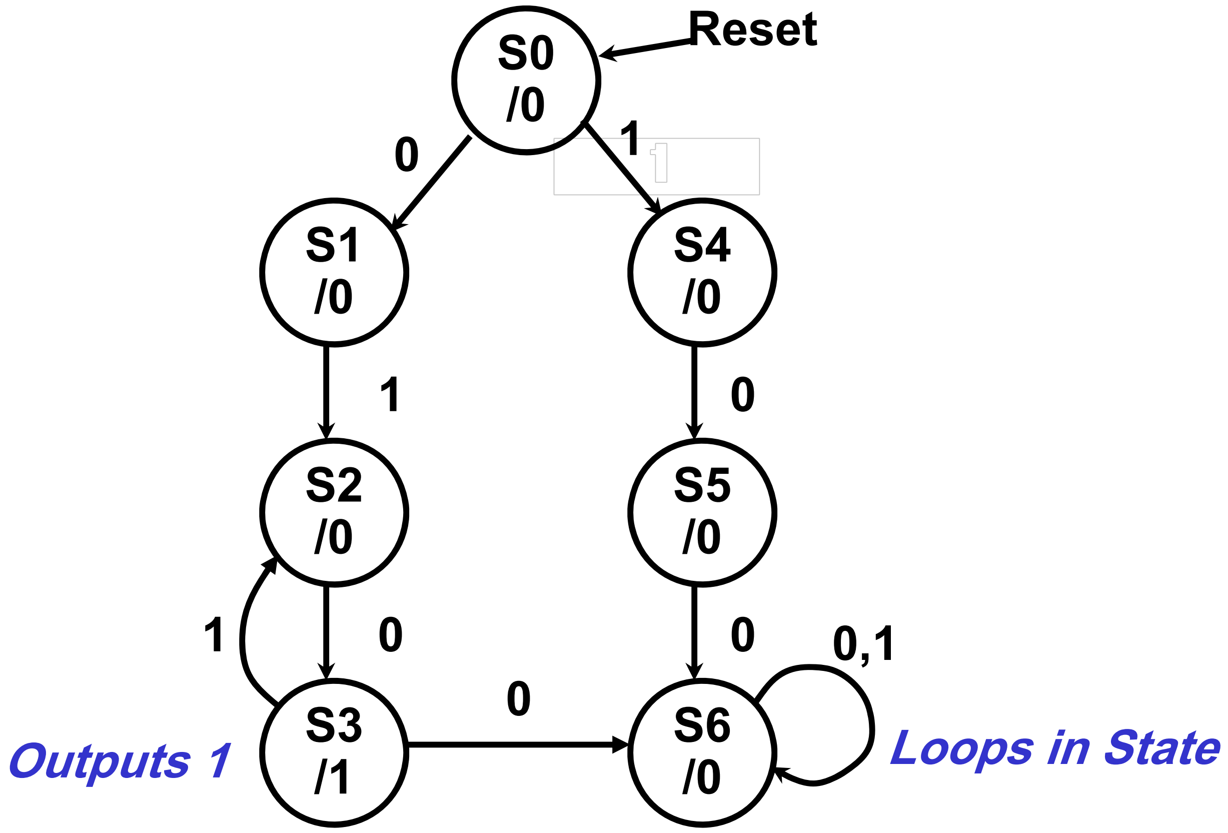
Outputs 1

Loops in State

Synchronous Sequential Circuit Word Problems

Finite String Recognizer

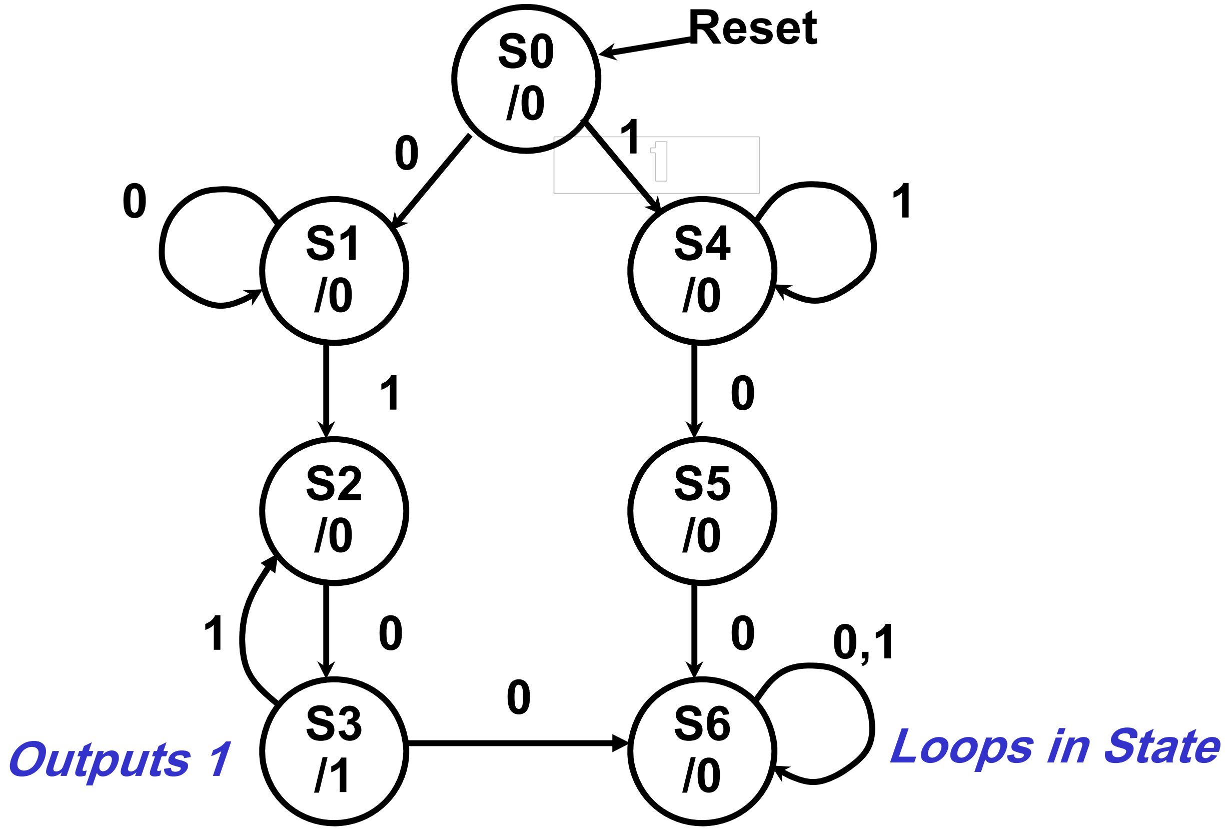
Exit conditions from state S3: have recognized …010
if next input is 0 then have …0100!
if next input is 1 then have …0101 = …01 (state S2)



Synchronous Sequential Circuit Word Problems

Finite String Recognizer

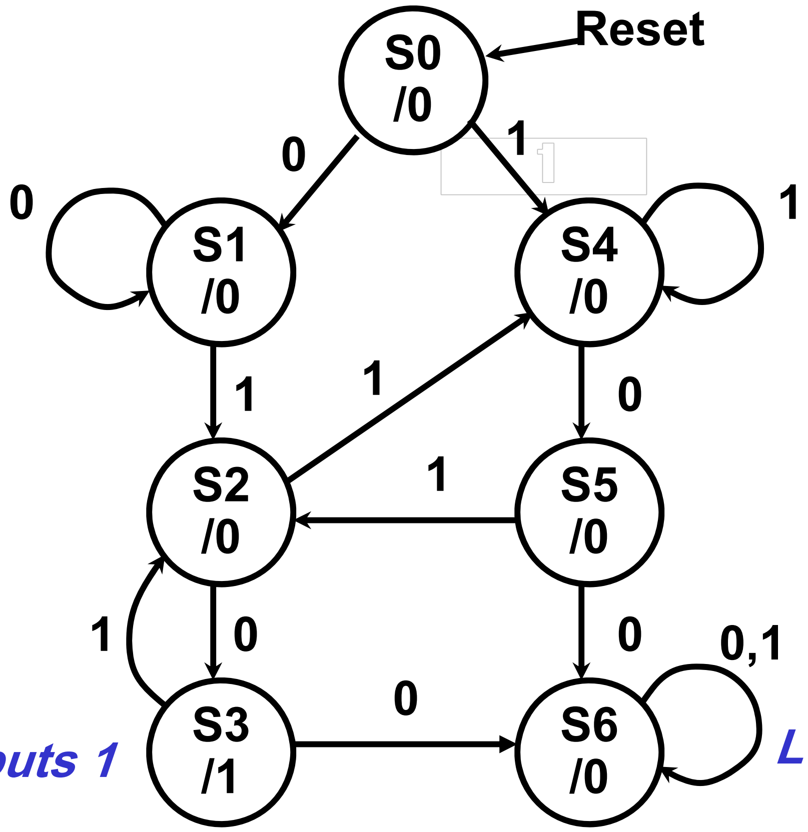
Exit conditions from S1: recognizes strings of form …0 (no 1 seen)
loop back to S1 if input is 0
Exit conditions from S4: recognizes strings of form …1 (no 0 seen)
loop back to S4 if input is 1



Synchronous Sequential Circuit Word Problems

Finite String Recognizer

S2 = …01; If next input is 1, then string could be prefix of (01)1(00)
S4 handles just this case!
S5 = …10; If next input is 1, then string could be prefix of (10)1(0)
S2 handles just this case!



Final State Diagram

Outputs 1

Loops in State

Synchronous Sequential Circuit Word Problems

Finite String Recognizer

Review of Process:

- **Write down sample inputs and outputs to understand specification**
- **Write down sequences of states and transitions for the sequences to be recognized**
- **Add missing transitions; reuse states as much as possible**
- **Verify I/O behavior of your state diagram to insure it functions like the specification**

Synchronous Sequential Circuit Word Problems

Complex Counter

A sync. 3 bit counter has a mode control M. When M = 0, the counter counts up in the binary sequence. When M = 1, the counter advances through the Gray code sequence.

Binary: 000, 001, 010, 011, 100, 101, 110, 111
Gray: 000, 001, 011, 010, 110, 111, 101, 100

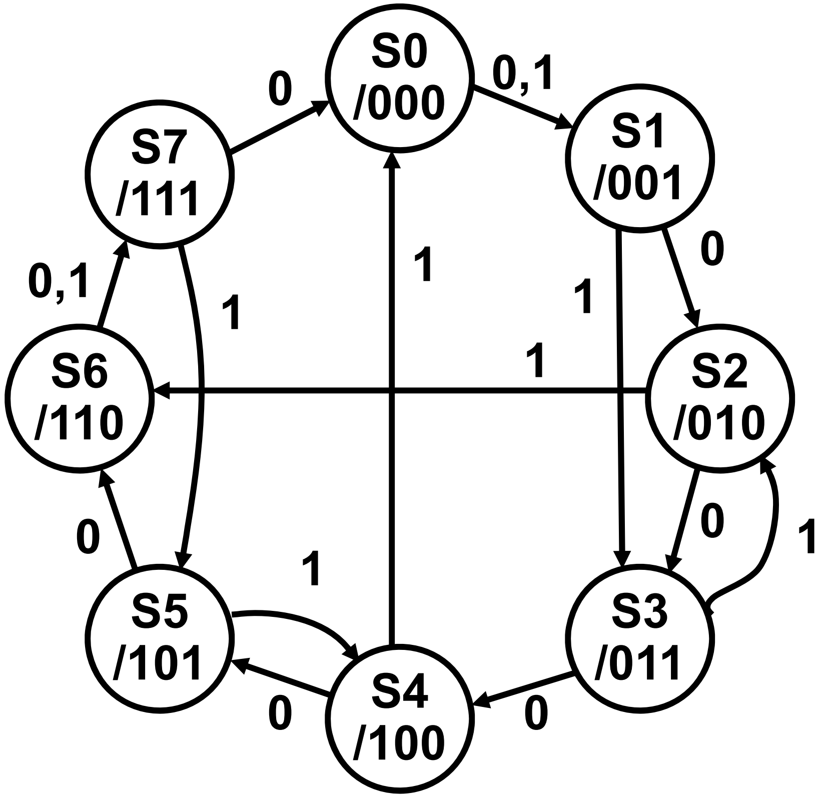
Valid I/O behavior:

Mode Input M	Current State	Next State (Z2 Z1 Z0)
0	000	001
0	001	010
1	010	110
1	110	111
1	111	101
0	101	110
0	110	111

Synchronous Sequential Circuit Word Problems

Complex Counter

One state for each output combination
Add appropriate arcs for the mode control



Synchronous Sequential Circuit Word Problems

Traffic Light Controller

A busy highway is intersected by a little used farmroad. Detectors C sense the presence of cars waiting on the farmroad. With no car on farmroad, light remain green in highway direction. If vehicle on farmroad, highway lights go from Green to Yellow to Red, allowing the farmroad lights to become green. These stay green only as long as a farmroad car is detected but never longer than a set interval. When these are met, farm lights transition from Green to Yellow to Red, allowing highway to return to green. Even if farmroad vehicles are waiting, highway gets at least a set interval as green.

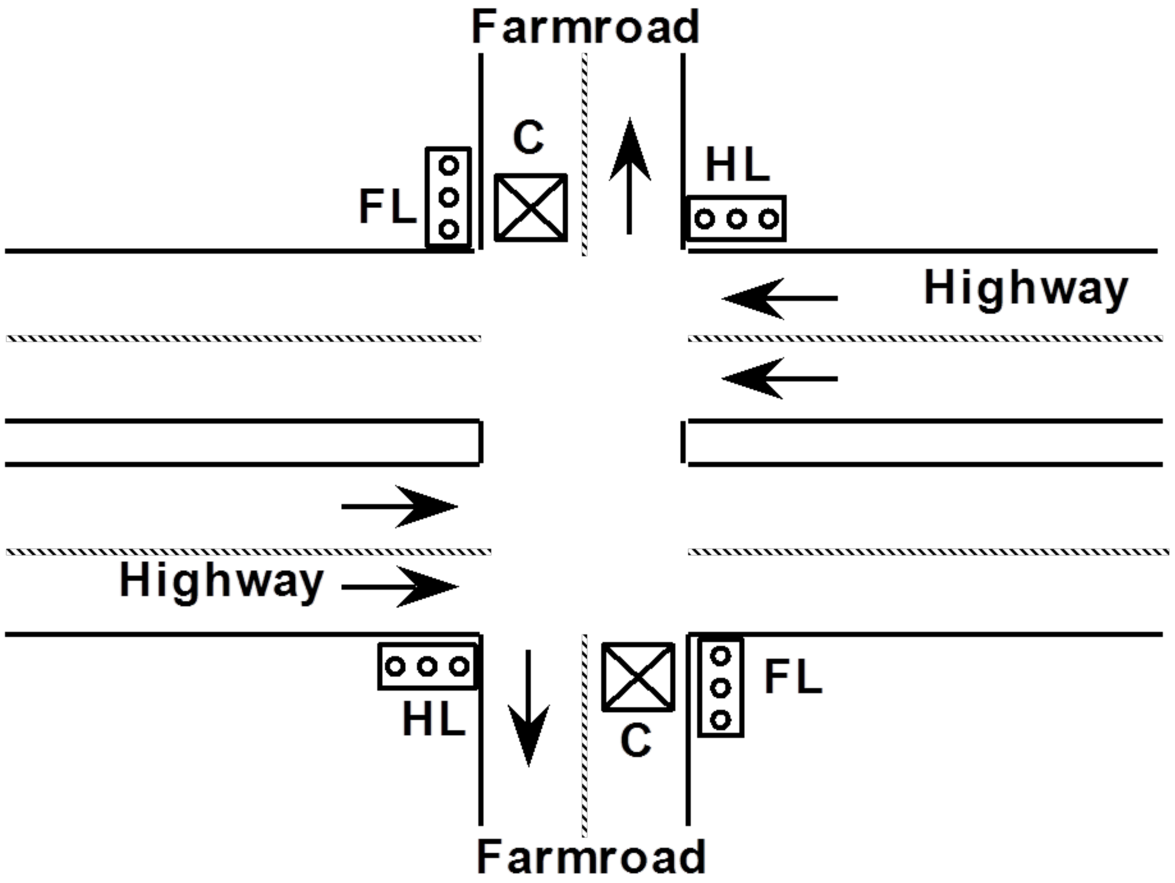
Assume you have an interval timer that generates a short time pulse (TS) and a long time pulse (TL) in response to a set (ST) signal. TS is to be used for timing yellow lights and TL for green lights.

Note: The interval timer is just another sequential circuit!

Synchronous Sequential Circuit Word Problems

Traffic Light Controller

Picture of Highway/Farmroad Intersection:



Synchronous Sequential Circuit Word Problems

Traffic Light Controller

- **Tabulation of Inputs and Outputs:**

Input Signal	***Description***
reset	**place SSC in initial state**
C	**detect vehicle on farmroad**
TS	**short time interval expired**
TL	**long time interval expired**

Output Signal	***Description***
HG, HY, HR	**assert green/yellow/red highway lights**
FG, FY, FR	**assert green/yellow/red farmroad lights**
ST	**start timing a short or long interval**

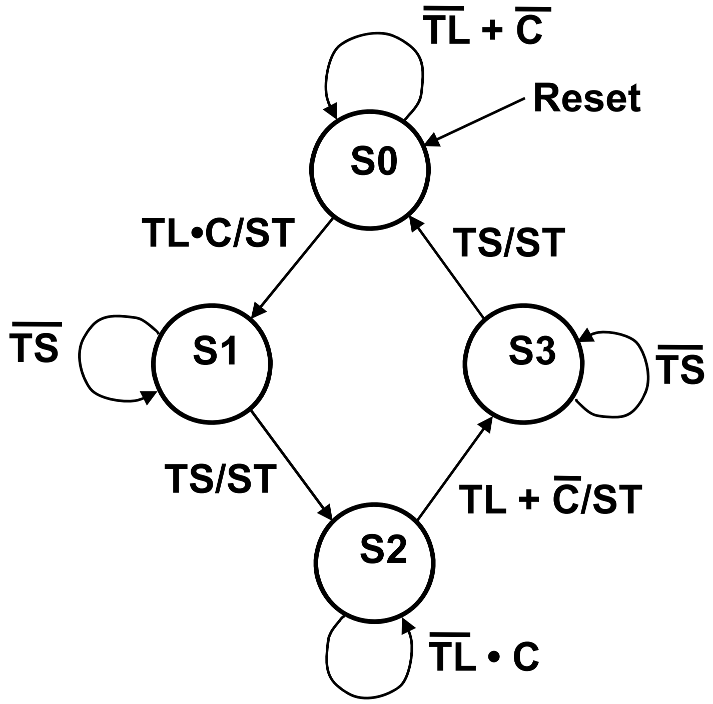
- **Tabulation of Unique States: Some light configuration imply others**

State	***Description***
S0	**Highway green (farmroad red)**
S1	**Highway yellow (farmroad red)**
S2	**Farmroad green (highway red)**
S3	**Farmroad yellow (highway red)**

Synchronous Sequential Circuit Word Problems

Traffic Light Controller

Compare with state diagram:



S0: HG, FR

S1: HY, FR

S2: FG, HR

S3: FY, HR

Note: This sequential circuit has both Mealy and Moore outputs!

Synchronous Sequential Circuit Word Problems

Digital Combination Lock

"3 bit serial lock controls entry to locked room. Inputs are RESET, ENTER, 2 position switch for bit of key data. Locks generates an UNLOCK signal when key matches internal combination. ERROR light illuminated if key does not match combination. Sequence is: (1) Press RESET, (2) enter key bit, (3) Press ENTER, (4) repeat (2) & (3) two more times."

Problem specification is incomplete:

- **how do you set the internal combination?**
- **exactly when is the ERROR light asserted?**

Make reasonable assumptions:

- **hardwired into next state logic vs. stored in internal register**
- **assert as soon as error is detected vs. wait until full combination has been entered**

Our design: registered combination plus error after full combination

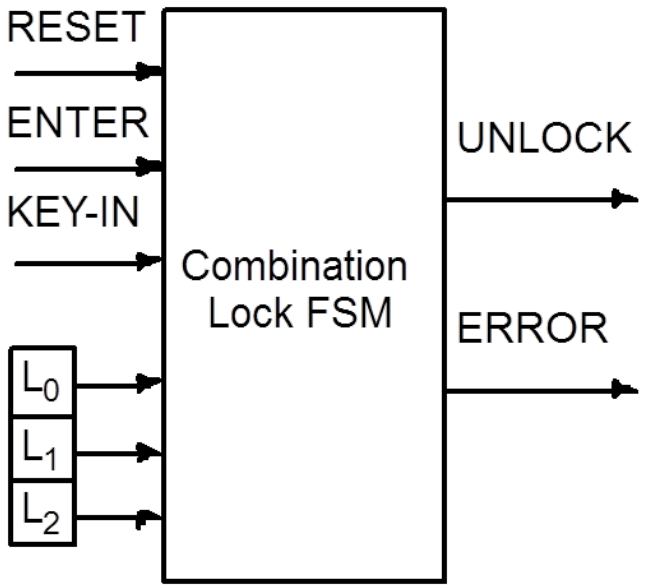
Synchronous Sequential Circuit Word Problems

Digital Combination Lock

Understanding the problem: draw a block diagram …

Operator Data

Internal Combination



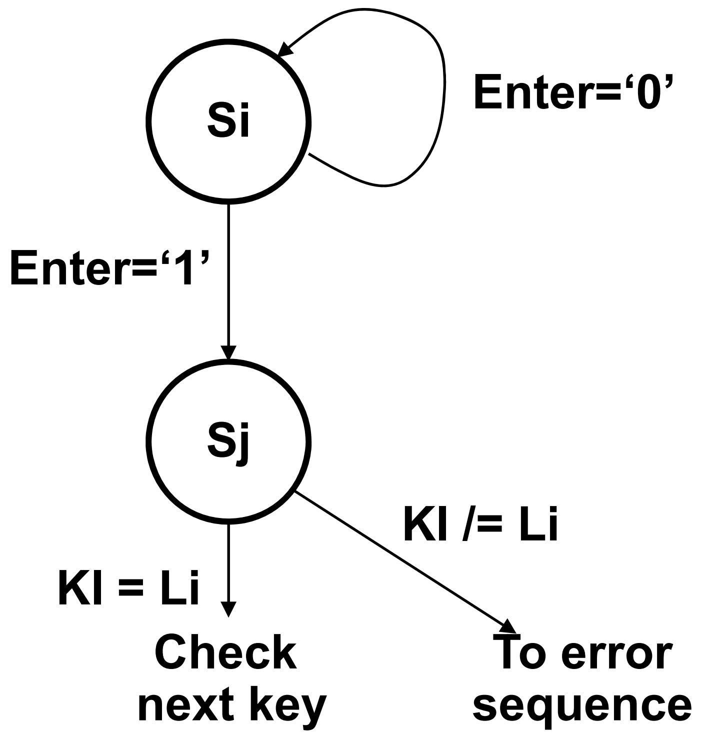
Inputs:
Reset
Enter
Key-In
L0, L1, L2

Outputs:
Unlock
Error

Synchronous Sequential Circuit Word Problems

Note that each key entry is really a two-step process

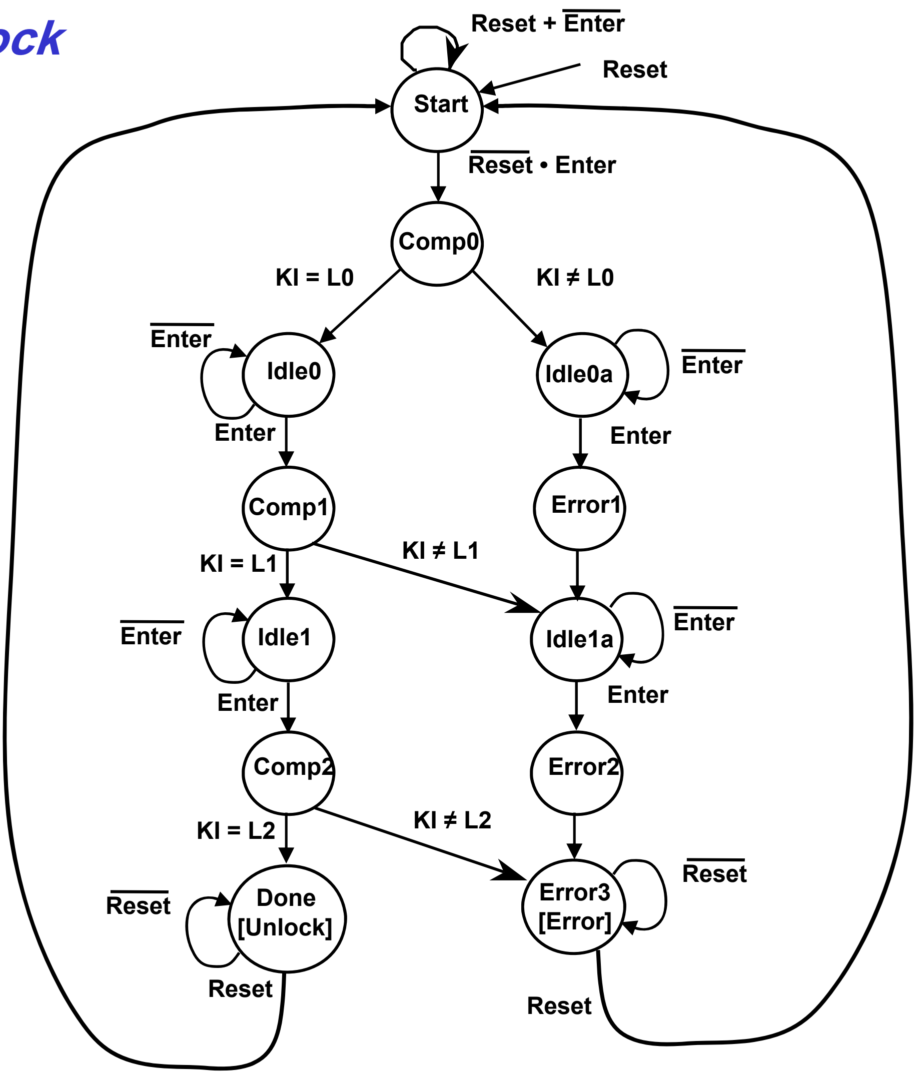
1. **Wait for the enter key**
2. **Check if correct key was selected**



Synchronous Sequential Circuit Word Problems

Digital Combination Lock

State Diagram



Reset + $\overline{\text{Enter}}$
Reset
Start
$\overline{\text{Reset}}$ • Enter
Comp0
Kl = L0
Kl ≠ L0
$\overline{\text{Enter}}$
Idle0
Idle0a
$\overline{\text{Enter}}$
Enter
Enter
Comp1
Error1
Kl ≠ L1
Kl = L1
$\overline{\text{Enter}}$
Idle1
Idle1a
$\overline{\text{Enter}}$
Enter
Enter
Comp2
Error2
Kl ≠ L2
Kl = L2
$\overline{\text{Reset}}$
Done [Unlock]
Error3 [Error]
$\overline{\text{Reset}}$
Reset
Reset