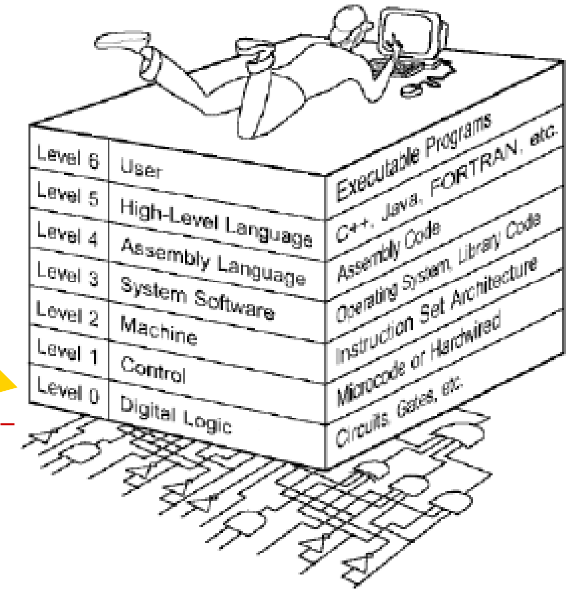
Human/natural language (Decimal numbers, letters & special characters) $\xrightarrow{\text{Ch-2}}$ Machine language (Binary/digital form)

binary
2's complement
Floating-point
ASCII, Unicode

Ch3 – Boolean Algebra & Digital Logic



LO-2:

-Use **Boolean algebra** mathematical expressions and k-maps to:

- **Describe** and manipulate the functions of simple combinational and sequential **logic circuits**.
- **Design** simple **combinational and sequential logic circuits** using gates and flip-flops.

>>> Quiz-2 and **Test-2**

Application of Boolean Algebra

- Digital circuit
- Google search
- Database (SQL)
- Programming
- ......

Simplification:

```
while  (((A && B) || (A && !B)) || !A)
{
// do something
}
```

$= AB+AB'+A'$
$= A(B+B')+A'$
$= A(1) + A'$
$= 1$

Boolean Algebraic simplifications helps to design efficient digital electronics circuits, computer programs, logical implementations, etc.

3.2 Boolean Algebra

- A mathematical system for manipulating binary variables with values:
  - "true" or "false" in formal logic
  - "on"/"off," "high"/"low," or "1"/"0" in digital systems
- Boolean operator function can be completely described using a ***Truth Table***.
- A Boolean function has at least each of Boolean variable, Boolean operator, and input from the set of {0,1}.
- Produces an output from set {0,1} – Either 0 or 1.

Boolean Product '.'

X AND Y

X	Y	XY
0	0	0
0	1	0
1	0	0
1	1	1

Boolean Sum '+'

X OR Y

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

Boolean Negation overbar (‾) or Prime (')

NOT X

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

Rules Of Precedence

- Arithmetic has its rules of precedence
  - Like arithmetic, Boolean operations follow the rules of precedence (priority):
  - NOT operator > AND operator > OR operator .
- This explains why we chose the shaded partial function in that order in the table.

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

x	y	z	$\bar{z}$	$x\bar{z}$	$x\bar{z}+y$
0	0	0	1	0	0
0	0	1	0	0	0
0	1	0	1	0	1
0	1	1	0	0	1
1	0	0	1	1	1
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	0	0	1

Rules Of Precedence

Use Boolean Algebra in Circuit Design

- Digital circuit designer always like achieve the following goals:
  - ***Cheaper*** to produce
  - Consume ***less power***
  - run ***faster***

Type text here

- How to do it? -- We know that:
  - Computers contain circuits that implement Boolean functions → Boolean functions can express circuits
  - If we can simplify a Boolean function, that express a circuit, we can archive the above goals
- We always can reduce a Boolean function to its *simplest* form by using a number of Boolean laws can help us do so.

Boolean Algebra Laws

Summary/cheat-sheet: all in one page

Identity Name	AND Form	OR Form
Identity Law	$1x = x$	$0 + x = x$
Null (or Dominance) Law	$0x = 0$	$1 + x = 1$
Idempotent Law	$xx = x$	$x + x = x$
Inverse Law	$xx' = 0$	$x + x' = 1$
Commutative Law	$xy = yx$	$x + y = y + x$
Associative Law	$(xy)z = x(yz)$	$(x + y) + z = x + (y + z)$
Distributive Law	$x + (yz) = (x + y)(x + z)$	$x(y + z) = xy + xz$
Absorption Law	$x(x + y) = x$	$x + xy = x$
DeMorgan's Law	$(xy)' = x' + y'$	$(x + y)' = x'y'$
Double Complement Law	$x'' = x$	

$X+X'Y = X+Y$

$(x+y)(x+z) = xx+xz+xy+yz$
$=x+xz+xy+yz = x(1+z)+xy+yz$
$= x+xy+yz = x(1+y)+yz$
$= x+yz$

LHS$=x(x+y)$
$= xx + xy$
$= x + xy$
$= x.1 + x.y$
$= x.(1+y)$
$= x.1$
$= x =$ RHS

Most important ones:

Distributive: $x+yz=(x+y)(x+z)$
DeMorgan's: $(xy)' = x'+y'$
x XOR y $= x'y + xy'$
$= (x+y)(x'+y')$

DeMorgan’s Law

- DeMorgan’s law can be extended to any number of variables.
  - Replace each variable by its negation (complement)
  - Change all ANDs to ORs and all ORs to ANDs.
- Let’s say F (X, Y, Z) is the following, what is $\bar{F}$?

$$F(X, Y, Z) = (XY) + (\overline{X}Y) + (X\overline{Z})$$

$$\begin{aligned} F' &= (X'+Y')(X+Y')(X'+Z) \\ &= (XX'+Y')(X'+Z) \\ &= X'Y'+Y'Z \end{aligned}$$

Logic simplification steps

- Apply De Morgan's theorems
- Expanding out parenthesis
- Find the common factors
- Popular rules used:

$X+XY=X$ $\quad$ $X+X=X$, $XX=X$

$XY+X\overline{Y}=X$ $\quad$ $X+0=X$, $X+1=1$

$X+\overline{X}Y=X+Y$ $\quad$ $X0=0$ $X1=X$

Simplifying Boolean Functions

- Let’s use Boolean laws to simplify:

as follows: $F(X,Y,Z) = (X+Y)\ (X+\overline{Y})\ (\overline{X\overline{Z}})$

$(X + Y)(X + \overline{Y})(\overline{X\overline{Z}})$	
$(X + Y)(X + \overline{Y})(\overline{X} + Z)$	DeMorgan's Law
	Double complement Law
$(XX + X\overline{Y} + YX + Y\overline{Y})(\overline{X} + Z)$	Distributive Law
$((X + Y\overline{Y}) + X(Y + \overline{Y}))(\overline{X} + Z)$	Commutative and Distributive Laws
$((X + 0) + X(1))(\overline{X} + Z)$	Inverse Law
$X(\overline{X} + Z)$	Idempotent and Identity Laws
$X\overline{X} + XZ$	Distributive Law
$0 + XZ$	Inverse Law
XZ	Identity Law

$$\overline{(xy)} = \bar{x} + \bar{y} \qquad \overline{(x+y)} = \bar{x}\,\bar{y}$$

Example (1)

- Apply De Morgan’s theorem

$\overline{WXYZ}$ = W' + X' + Y' + Z'

$\overline{W+X+Y+Z}$ = W' X' Y' Z'

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

$\overline{A\bar{B}+\bar{C}D+EF}$ = (A'+B) (C+D') (E'+F')

Example (2)

- $(A\overline{B}(C+BD)+\overline{A}\,\overline{B})C$

= (AB'C + AB'BD + A'B') C
= AB'CC + 0 + A'B'C
= (A+A') B'C
= B'C

Example (3)

- $\overline{A}BC+A\overline{B}\,\overline{C}+\overline{A}\,\overline{B}\,\overline{C}+A\overline{B}C+ABC$

=A'BC+B'C'(A+A')+AC(B+B')
=A'BC+B'C'+AC
=(A'B+A)C+B'C'
=(A'+A)(B+A)C+B'C'   [Distributive Law]
= AC+BC+B'C'

Example (4)

- $\overline{(AB+AC)}+\overline{A}\,\overline{B}C$

= (A'+B')(A'+C')+A'B'C
= A'+A'C' +A'B'+A'B'C   +B'C'
= A' (1+C') + A'B' (1+C) +B'C'
= A'+ A'B' + B'C'   Null law
= A'+B'C'   Null law
= (A(B+C))'

Example (5)

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

- $A\overline{C}+A\overline{B}C+ABCD+AB\overline{D}$

= A[(C'+CB') + B(D'+DC)]
= A[(C'+B') + B(D'+C)]
= A[C'+CB + B'+BD']
=A[C'+B + B'+D']
=A[C'+1 + D']
=A[1 + D']
=A1
= A

Example (6)

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

- $(\overline{A}+\overline{B}+\overline{C})(\overline{B}+C)(A+\overline{B})$

= (A'+B'+C') (AB'+AC+B'+B'C)
= (A'+B'+C') (AC+AB'+B')
= (A'+B'+C') (AC+B') [Distributive Law]
= 0+A'B'+AB'C+   B'+0+B'C'
= B' (A'+AC +   1+ C')
= B' (A'+C +   1)
= B' (A'+ 1)
= B'

Revisiting and Simplifying the Example on Programming

```
while  (((A && B) || (A && !B)) || !A)
{
        // do something
}
```

=

```
while (1)
{
       // do something
}
```

$= AB+AB'+A'$
$= A(B+B')+A'$
$= A(1) + A'$
$= 1$

Boolean Algebra: Standardization

- Through our exercises in simplifying Boolean expressions, we see that there are **more than one ways to express the same Boolean function**.
  - These "synonymous" forms are *logically equivalent.*
  - Logically equivalent expressions could produce confusions

$$(X+Y)(X+\overline{Y})(\overline{X\overline{Z}}) = XZ$$

- In order **to eliminate the confusion**, designers express Boolean expression in a unified and *standardized* form, called ***canonical form***.

Boolean Algebra: Minterm and Maxterm

1D Truth Table

X AND Y		
X	Y	XY
0	0	0
0	1	0
1	0	0
1	1	1

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

- Some books uses *sum-of-minterms form* and *product-of-maxterms form*
  - A *minterm* is a logical expression of n variables that employs only the complement operator and the product operator.
    - For example, **abc, ab'c** and **abc'** are 3 minterms for a Boolean function of the three variables a, b, and c.
  - A *maxterm* is a logical expression of n variables that employs only the complement operator and the sum operator.

Each variable appears only once in each minterm. Only one minterm based on all variables will be ON/1. Conversely, all but one terms will be OFF/0.

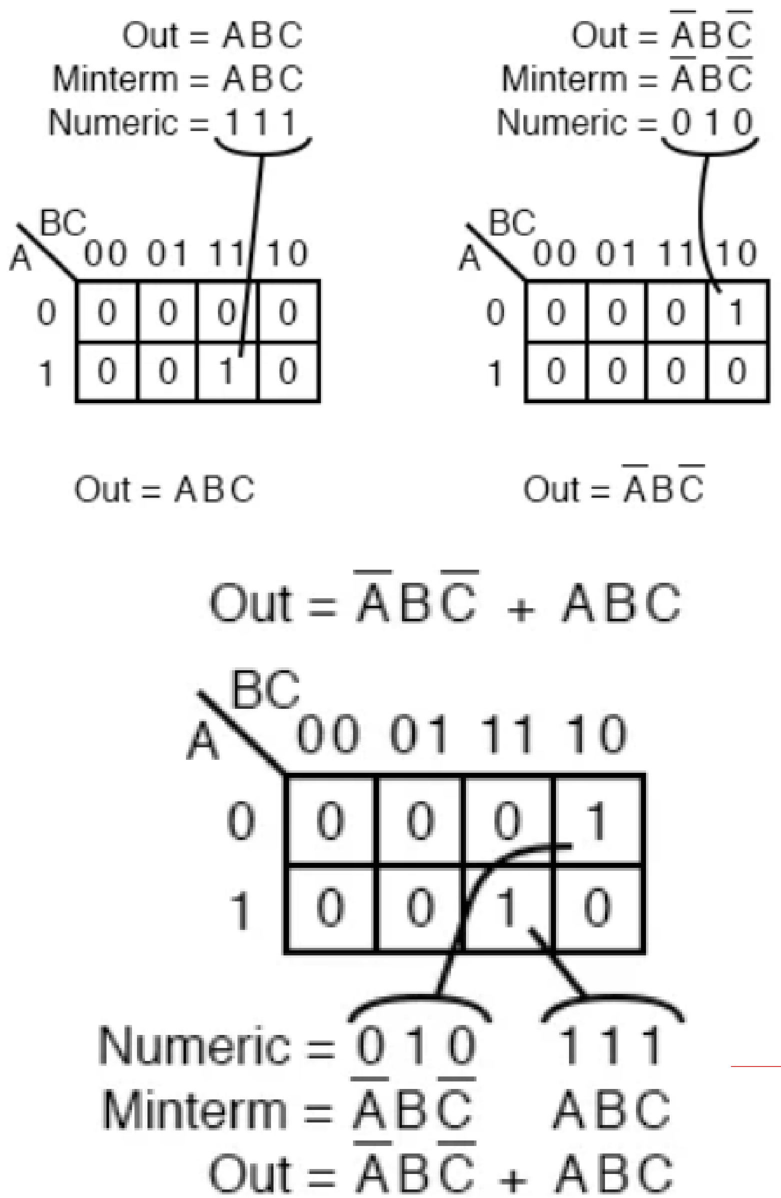
Each variable appears only once in each maxterm. Only one maxterm based on all variables will be OFF/0. Conversely, all but one terms will be ON/1.

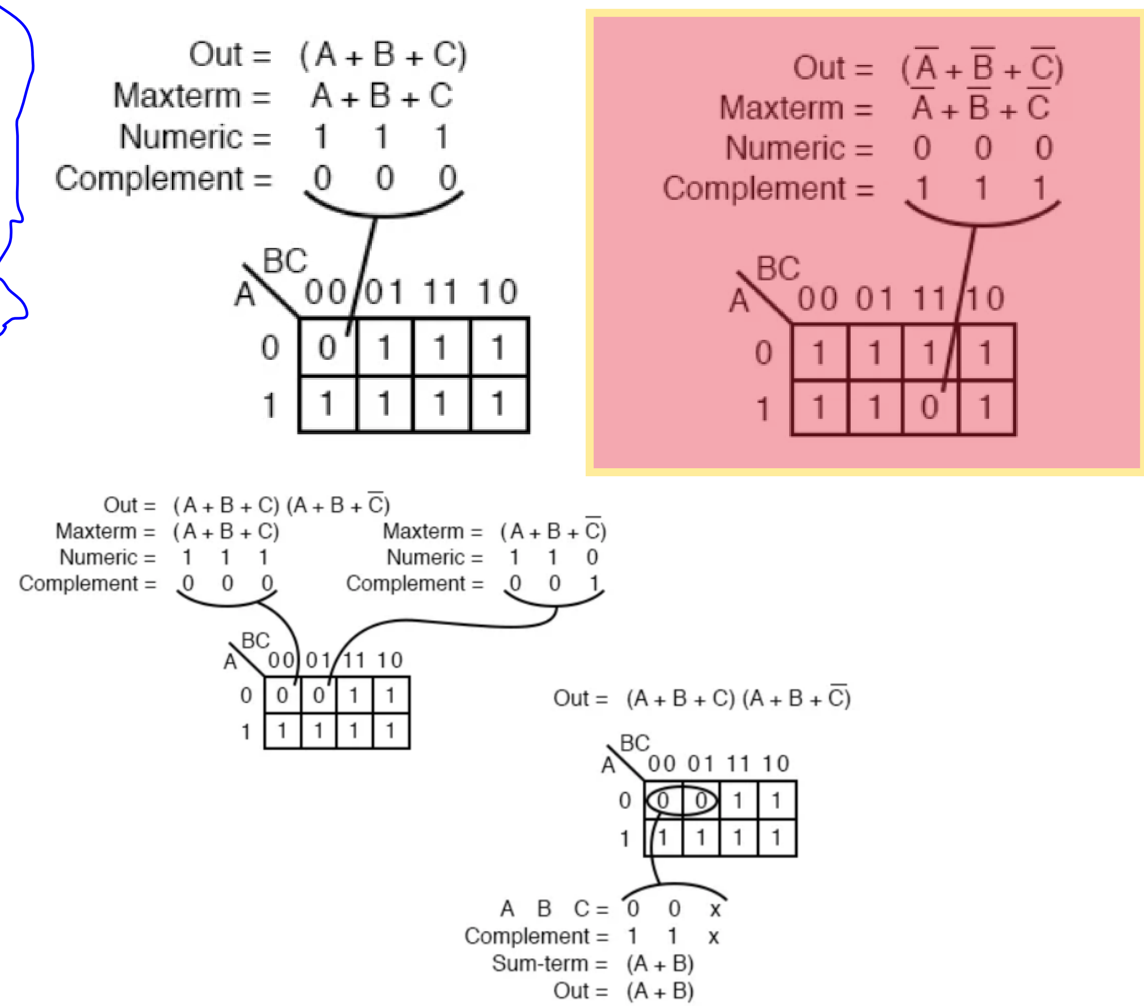
2D Truth Table

x+y

x\y	0	1
0	0	1
1	1	1

Logic Reduction: Minterm / Maxterm - Example using Karnaugh (K) map





https://www.allaboutcircuits.com/textbook/digital/chpt-8/minterm-maxterm-solution/

Boolean Algebra: Standardization

- There are two canonical forms for Boolean expressions: sum-of-products and product-of-sums.
  - Boolean product (x) →***AND*** →logical conjunction operator
  - Boolean sum (+) → ***OR*** →logical conjunction operator
- In the *sum-of-products form*, ANDed variables are *ORed together*.

  The example below is SOP but not SOm. Every minterm is a product, opposite may not be true. A simple form is non-unique. SOm help us to find minimal simple form which almost becomes a unique simple form.

  - For example: $F(x,y,z) = xy + xz + yz$
- In the *product-of-sums form*, ORed variables are *ANDed together*:

  The example below is POS but not POM. A maxterm is a sum, opposite may not be true. A simple form is non-unique. SOM help us to find minimal simple form which almost becomes a unique simple form.

  - For example: $F(x,y,z) = (x+y)(x+z)(y+z)$

Create Canonical Form Via Truth Table ('Cont)

- Look at this example:

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

$$= (\bar{x}y\bar{z}) + (\bar{x}yz) + (x\bar{y}\bar{z}) + (xy\bar{z}) + (xyz)$$

- It may not be the simplest form. But, it is the standard sum-of-products ***canonical form***

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

	x	y	z	$x\bar{z}+y$
	0	0	0	0
	0	0	1	0
$(\bar{x}y\bar{z})$	0	1	0	1
$(\bar{x}yz)$	0	1	1	1
$(x\bar{y}\bar{z})$	1	0	0	1
	1	0	1	0
$(xy\bar{z})$	1	1	0	1
(xyz)	1	1	1	1

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

Exercise

- **Convert** ABC+A'BC+AB'C+A'B'C+ABC' **to** its **simplest form**

ABC+A'BC+AB'C+A'B'C+ABC'= BC(A+A') + B'C(A+A') + ABC'
= BC1 +B'C1 + ABC'
= C(B+B') + ABC'
= C + ABC'
= C + AB

Distributivity
C+AB = (C+AB) (C+C')
=C+0+ABC+ABC' = C(1+AB)+ABC'
=C1+ABC' = C+ABC'

Exercise

- **<u>Convert</u>** AB + C **<u>to</u>** the **<u>sum-of-products form</u>**

AB	= AB 1	By Th4
=	AB (C + C')	By Th 15
=	ABC + ABC'	By distributive law
=	CBA + C'BA	By associative law

C	= C 1	By Th4
=	C (A + A')	By Th15
=	CA + CA'	By distributive law
=	CA 1 + CA'1	By Th4
=	CA (B + B') + CA' (B + B')	By Th15
=	CAB + CAB' + CA'B + CA'B'	By distributive law
=	CBA + CBA' + CB'A + CB'A'	By associative law

AB+C = (CBA + C'BA) + (CBA + CBA' + CB'A + CB'A')

= CBA + CBA' + CB'A + CB'A' + C'BA

ABC+A'BC+AB'C+A'B'C+ABC'

Converting b/w Canonical Forms: by examples

when you want o/p logic **high**

Alternate representations by Cx1

when you want o/p logic **low**

SOP/SOm

	x	y	F=xy'+x'y
	0	0	0
x'y	0	1	1
xy'	1	0	1
	1	1	0

using DeMorgan's Law

F'=(xy'+x'y)'=(x'+y)(x+y')

1) Complementing algebraic function

2) Complementing logic values

POS/POM

	x	y	F'=(x'+y)(x+y')
	0	0	1
(x+y')	0	1	0
(x'+y)	1	0	0
	1	1	1

x	y	z	F=xyz+xyz'+x'yz+x'yz'+xy'z+xy'z'
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

==> F=y+xy'=x+y

==> F'=x'y'=x'y'z'+x'y'z

==> F=(x'y'z'+x'y'z)'

x	y	z	F'=x'y'
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Dual representations by Cx2

SOP/SOm = (SOP ') '
=Complemented ("another") SOP

"another" SOP/SOm

when you want o/p logic **low**

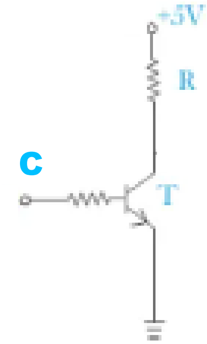
3.3 Digital Logic Gates

- We've seen Boolean functions in abstract terms.
- You may still ask:
  - ***How could Boolean function be used in computer?***
- In reality, Boolean functions are implemented as digital circuits, which called ***Logic Gates***.
- A logic gate is an electronic device that produces a result based on input values.
  - A logic gate may contain multiple transistors, but, we think them as one integrated unit.
  - **Integrated circuits** (IC) contain collections of gates, for a particular purpose.

Recap:

Transistor is an electronic switch!

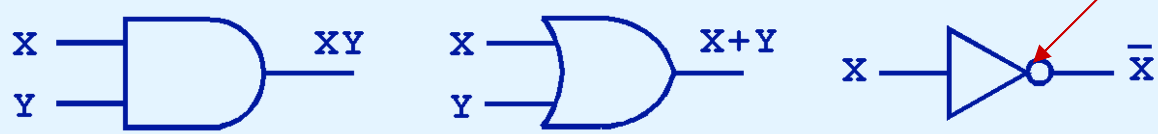
Normally open mechanical switch **pressed** = transistor switched **on** = current can flow

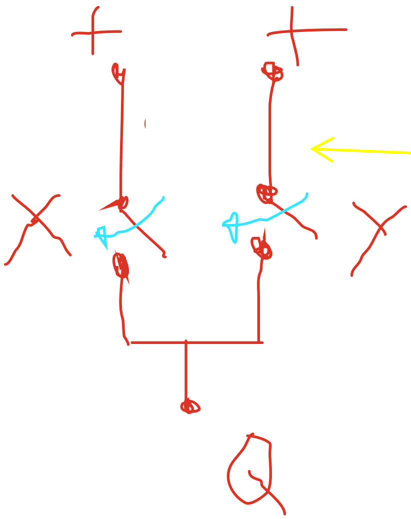


AND, OR, and NOT Gates

- Three simplest gates are the AND, OR, and NOT gates. Their symbol and truth tables are shown below.

"inversion bubble"





X AND Y

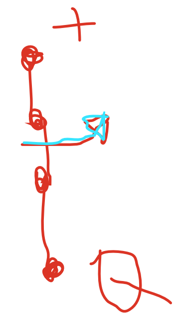
X	Y	XY
0	0	0
0	1	0
1	0	0
1	1	1

X OR Y

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

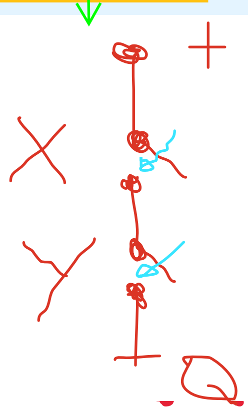
NOT X

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



Normally closed switch is used in NOT logic circuit implementation.

Normally open switches are used in AND and OR logic circuit implementations.



NAND/NOR Gates

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

SOP <--> POS

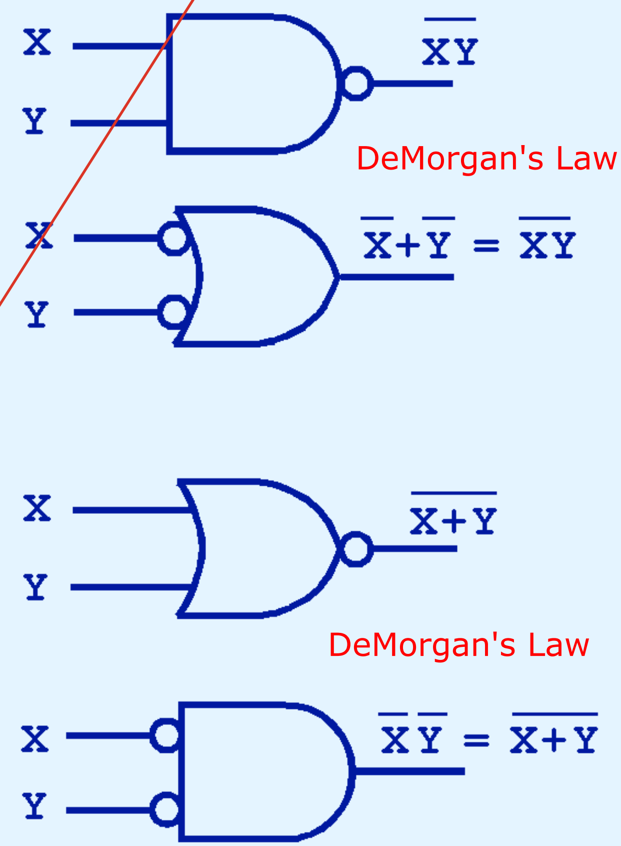
- NAND and NOR are two additional gates.
  - Their symbols and truth tables are shown on the right.
- NAND = NOT AND
- NOR = NOT OR

X NAND Y

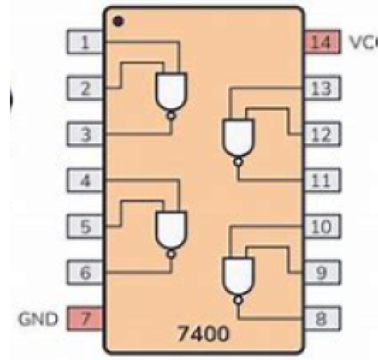
X	Y	X NAND Y
0	0	1
0	1	1
1	0	1
1	1	0

X NOR Y

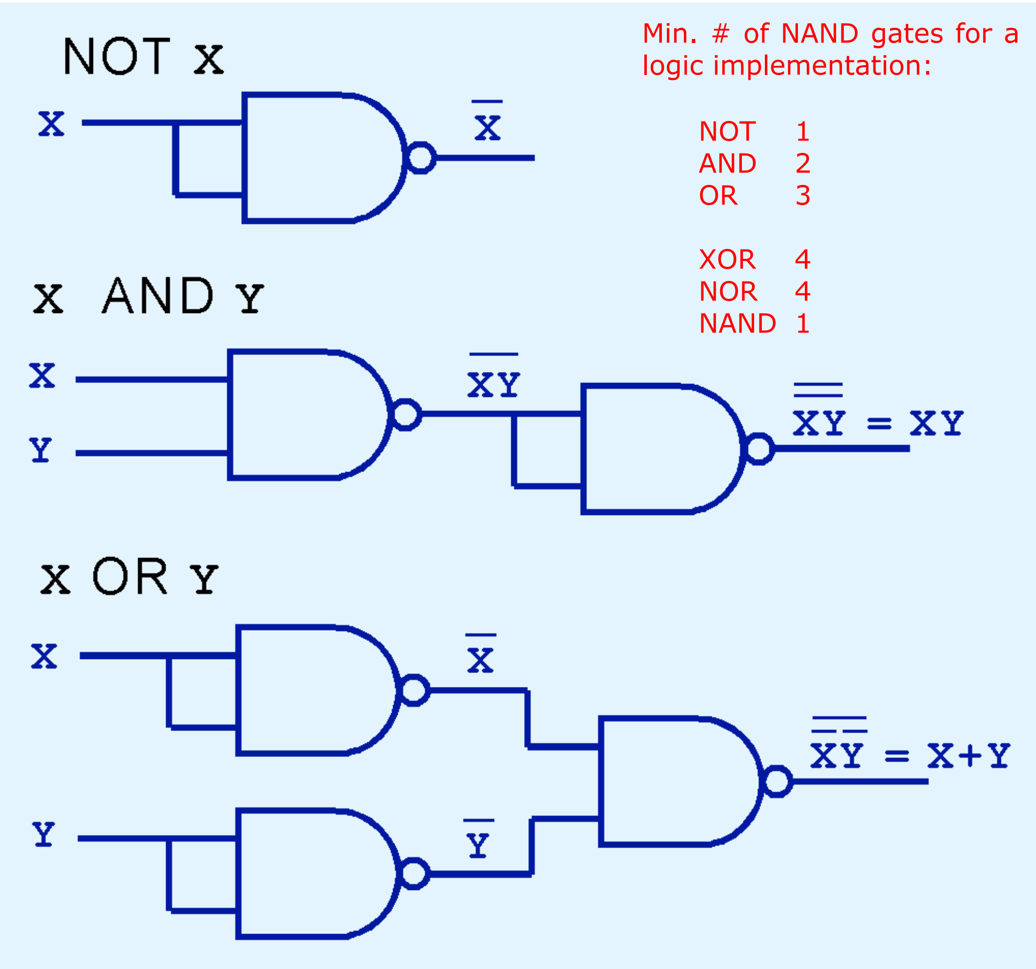
X	Y	X NOR Y
0	0	1
0	1	0
1	0	0
1	1	0



The Application of NAND/NOR Gates

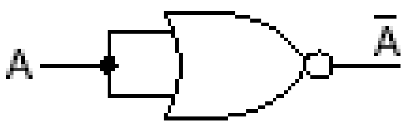


- NAND and NOR are known as [*universal gates! – gates of all gates*]
  - They are inexpensive to produce
- More important: Any Boolean function can be constructed using only NAND or only NOR gates.

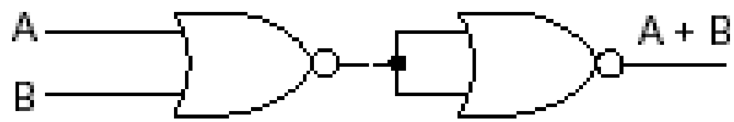


The Universal Gates: NOR Gate

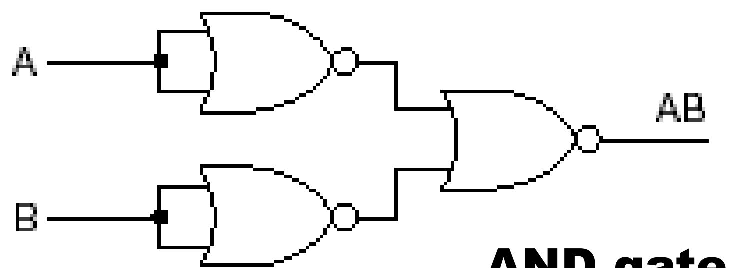
- Using ***NOR*** gate to construct AND, OR, and NOT gates



Not gate



OR gate

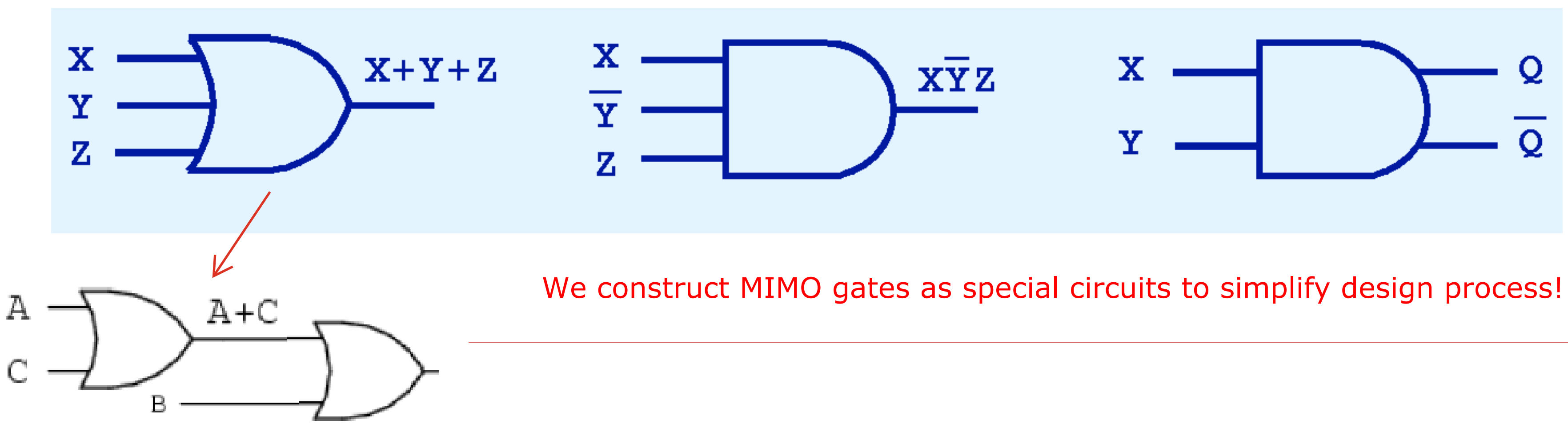


AND gate

logic	Min # NANDs	Min # NORs
NOT	1	1
AND	2	3
OR	3	2
XOR	4	4
NOR	4	1
NAND	1	4

Multiple Input/Output Gates

- The gates could have multiple inputs and/or multiple outputs.
  - The second output can be provided as the complement of the first output.
  - We'll see more integrated circuits, which have multiple inputs/outputs.



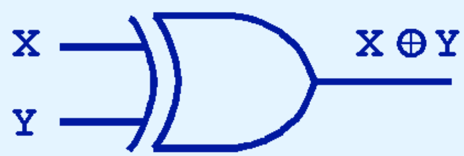
We construct MIMO gates as special circuits to simplify design process!

XOR Gates

- Another very useful gate is the ***Exclusive OR*** (XOR) gate.
- The output of the XOR operation is true (1) only when the values of inputs are different.

X XOR Y

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



x XOR y
$= x'y + xy'$
$= (x+y)(x'+y')$

Can you implement an XOR gate?
Try it!

- The symbol for XOR is $\oplus$

Parity generator / checker

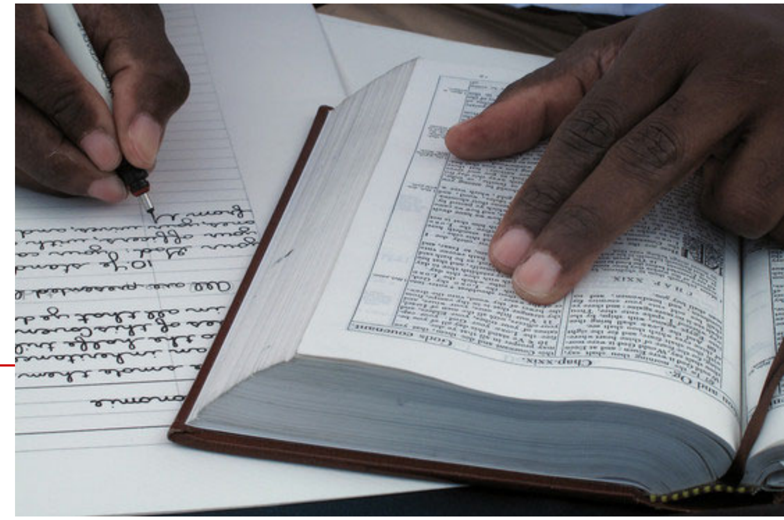
- Electrical **noise** in the transmission of binary information can **cause errors**
- **Parity can detect** these types of errors
- Parity systems
  - Odd parity
  - Even parity
- **Add a bit** to the binary information

Even parity check

- Even parity check
- Example: input: A(7…0), Output: even_parity bit
  - If there are even numbers of 1 in A, even_parity = ‘0’,
  - If there are odd numbers of 1 in A, even_parity = ‘1’

  e.g., A = “10100001”,
  even_parity = ‘1’
  A = “10100011”,
  even_parity = ‘0’

Odd parity check



- Odd parity check
- Example: input: A(7…0), Output: odd_parity bit
  - If there are odd numbers of 1 in A, odd_parity = ‘0’,
  - If there are even numbers of 1 in A, odd_parity = ‘1’

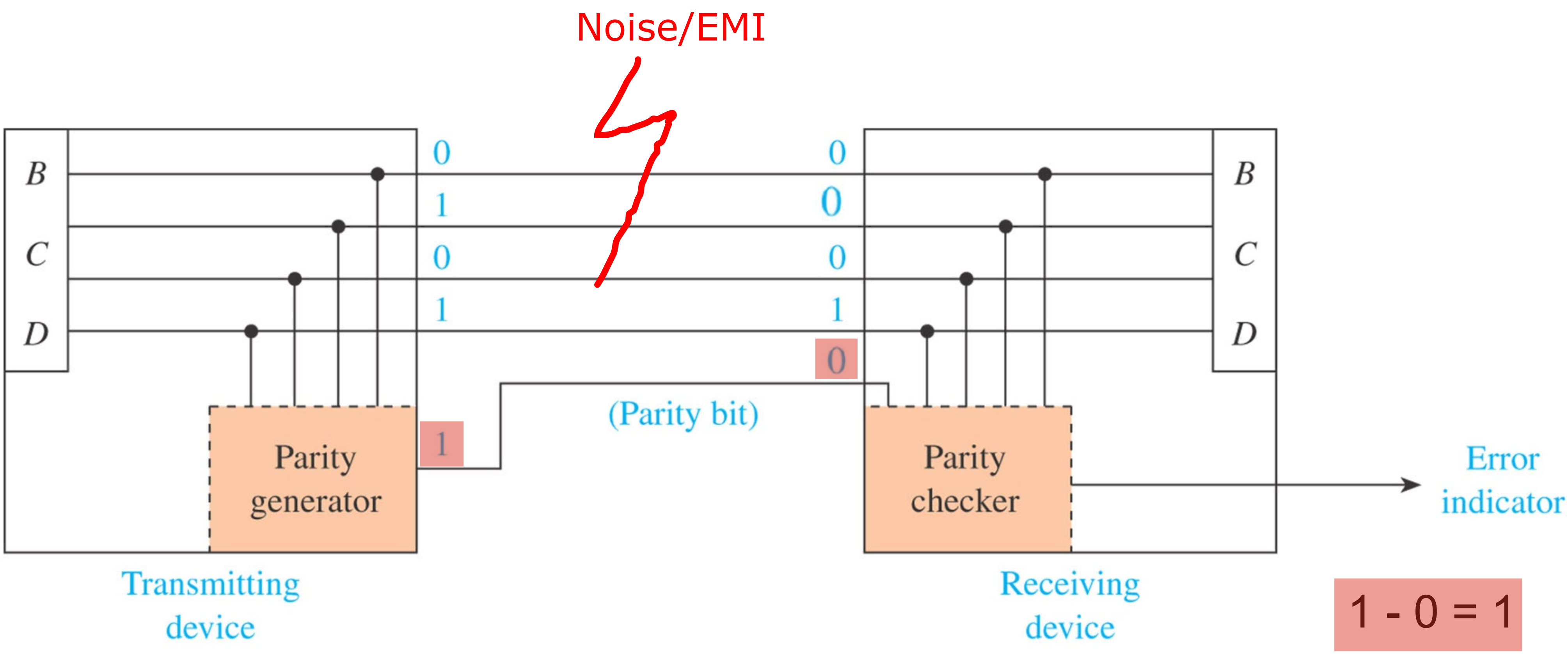
e.g., A = “10100001”,

odd_parity = ‘0’

A = “10100011”,

odd_parity = ‘1’

Odd-parity generator/checker system



Error detection

- Transmitting end: The parity generator creates the parity bit.
- Receiving end: The parity checker determines if the parity is correct.
- e.g., odd-parity check of 8-bit data
  - Data send: 10111101 + 1
  - Data received: 101011010

  odd-parity check: The number of 1's sent were even BUT received odd numbered → *ERROR*

Discussion point

- What are **disadvantages** of even parity (or odd parity) check to detect transmission errors? Consider the following case:
  - Protocol: 8-bit plus one even parity bit
  - Information sent: $\boxed{11011100}$ + 1
  - Information received: $\boxed{10010100}$ + 1
- The parity generator/checker system **detects only one error** that occur to 1 bit.

Parity check using XOR

- ***N*-1 XOR gates** can be **cascaded** to form a circuit with ***N* inputs** and a single output
  - *even-parity circuit.*
    - Example: N=8, Inputs=10111101, *even-parity output*

      $=(\overset{1}{(\overset{1}{(1\oplus 0)}\oplus\overset{0}{(1\oplus 1)})})\oplus(\overset{1}{(\overset{0}{(1\oplus 1)}\oplus\overset{1}{(0\oplus 1)})})=0$
- **Odd-parity check** circuit: ***even-parity*** *check circuit* ➔**Inverted**➔Odd-parity check
  - Example: N=8, Inputs=10111101, odd-*parity output*

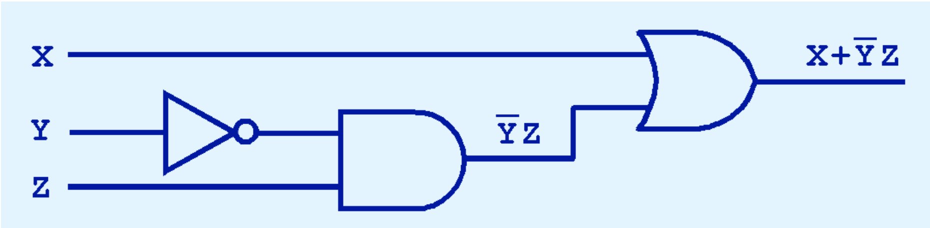
    $=\mathrm{NOT}(((1\oplus 0)\oplus(1\oplus 1))\oplus((1\oplus 1)\oplus(0\oplus 1)))=1$

Two Types of Logic Circuits

- Combinational Logic Circuit (*CLC*)
  - *Good at designing computational components in the CPU, such as ALU*
- Sequential Logic Circuit (*SLC*)
  - *Good at designing memory components, such as registers and memory*

as well as Control Unit (CU) and State Machines

3.5 Combinational Circuits



- The circuit implements the Boolean function:

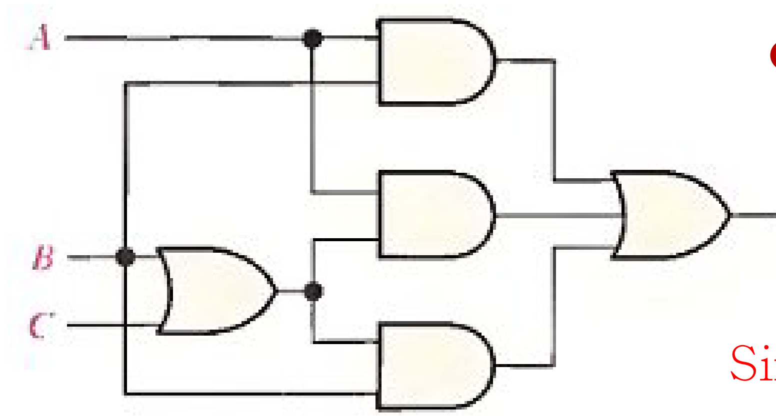
$$F(X,Y,Z) = X+\overline{Y}Z$$

- The major characteristics of this kind of circuits:
  - ***The circuit*** produces an output almost immediately after the inputs are given.
- This kind of circuits are called ***combinational logic circuit (CLC)***.
  - In a later section, we will explore circuits where this is not the case.

Simplify *CLC via* Bool
Algebra

Can we simplify this circuit? If yes, then how?

- Look at this example

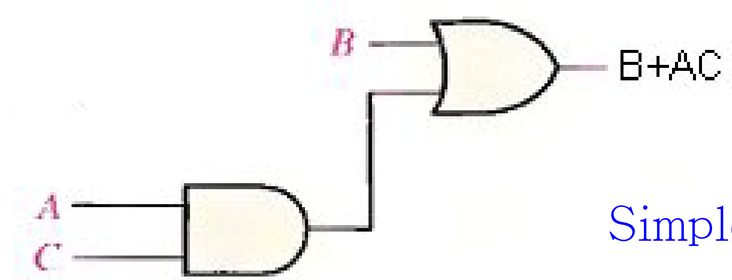


Express

$$AB + A(B + C) + B(B + C)$$
$$= AB + AB + AC + BB + BC$$
$$= AB + AB + AC + B + BC$$
$$= AB + AC + B + BC$$
$$= AB + AC + B$$
$$= B + AC$$

Simplify

Redraw

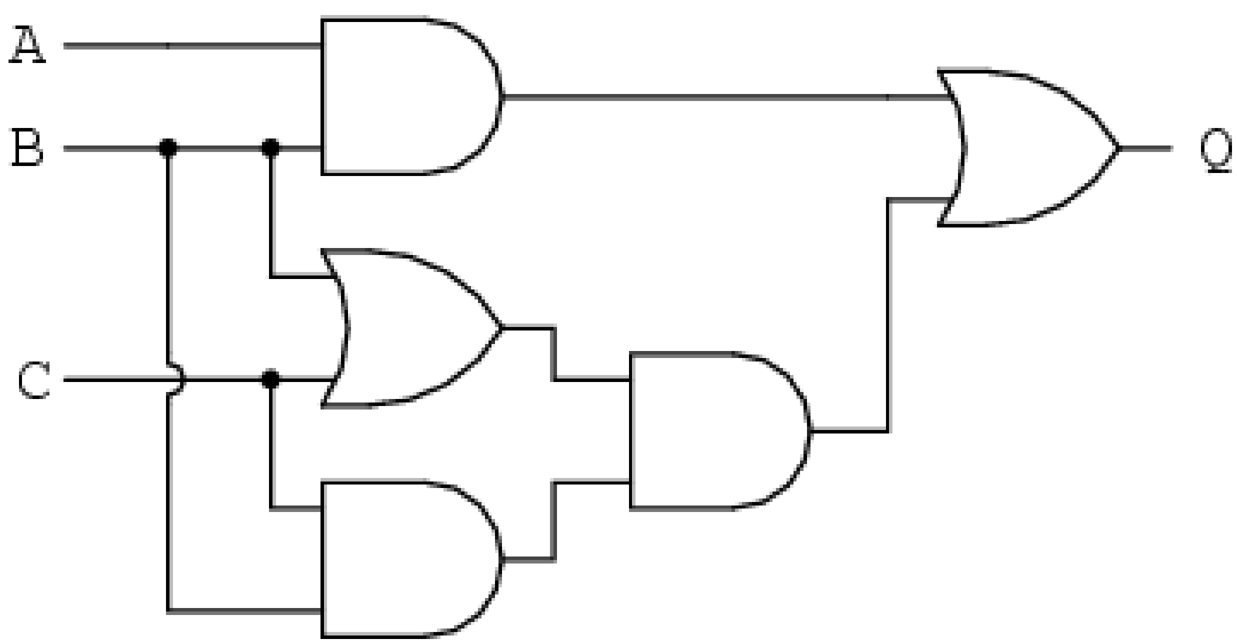


Simpler Boolean Function --> Simple Digital Circuit

Simpler circuits are cheaper →consume less power →run faster than complex circuits.

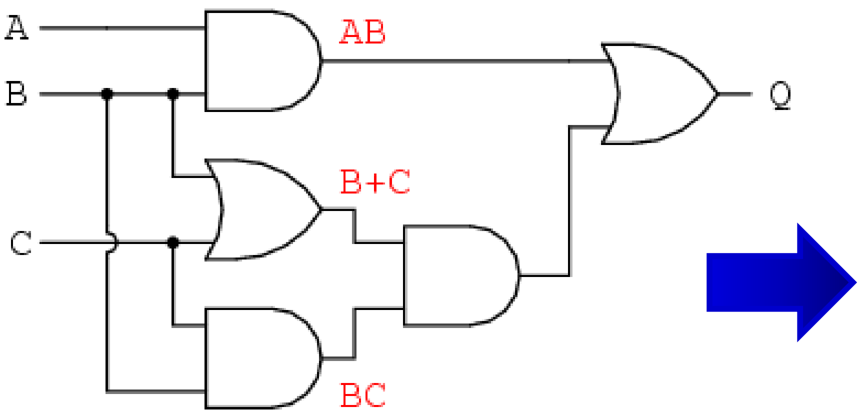
Example of Simplify a Logical Circuit

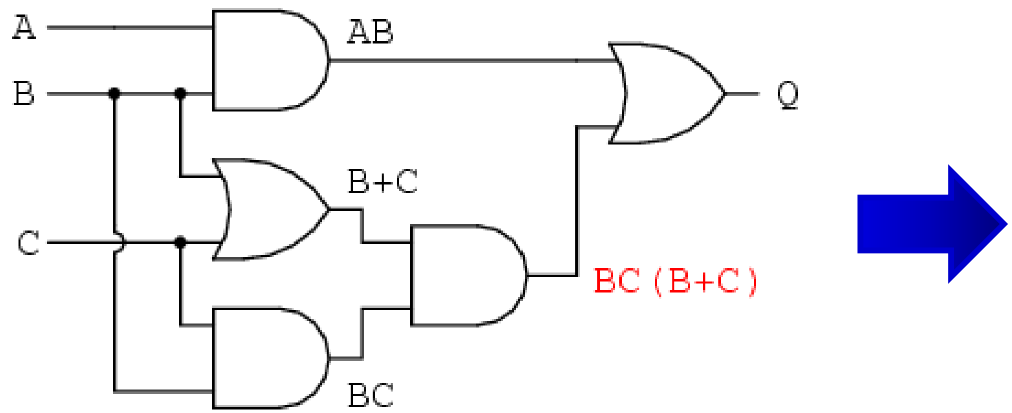
- Simplify the following circuit

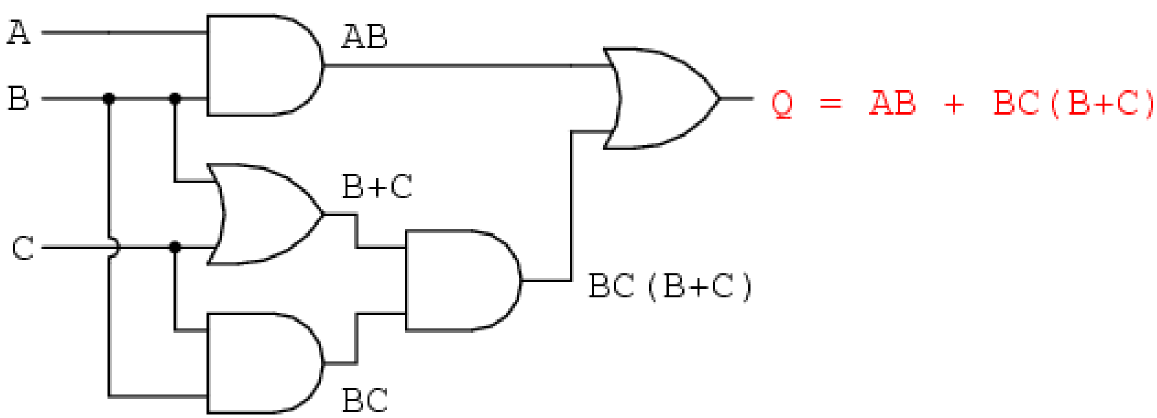


Example of Simplify a Logical Circuit

- Step1: Express a logical circuit into a Boolean expression







Example of Simplify a Logical Circuit

- Step2: Simplify the Boolean expression as much as possible

$$AB + BC(B + C)$$

$\downarrow$ Distributing terms

$$AB + BBC + BCC$$

$\downarrow$ Applying identity $AA = A$ to 2nd and 3rd terms

$$AB + BC + BC$$

$\downarrow$ Applying identity $A + A = A$ to 2nd and 3rd terms

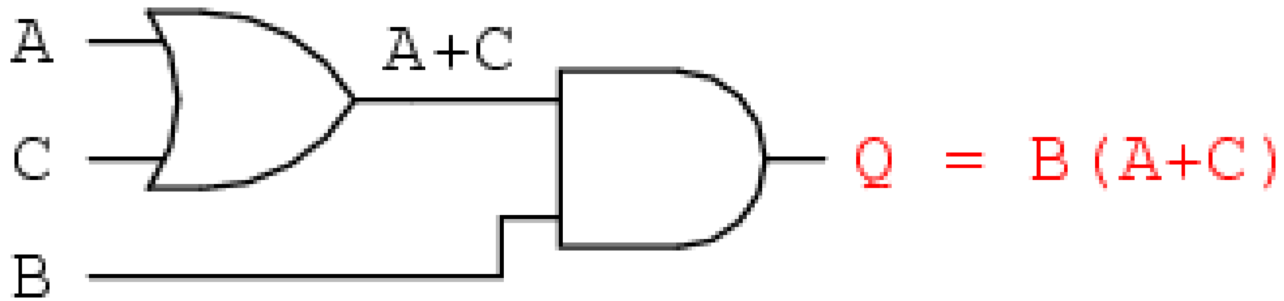
$$AB + BC$$

$\downarrow$ Factoring B out of terms

$$B(A + C)$$

Example of Simplify a Logical Circuit

- Step3: Re-express the simplified expression back to a circuit



Obviously, the simplified circuit is much *simpler* than the original one

Combinational Circuits: *Half (bit) Adder*

two binary numbers
(2-bit system)
X = X1 Xo
Y = Y1 Yo

- Combinational logic circuits can be used to create many useful devices.
- *Half Adder*: Compute the sum of two bits.
- Let's gain some insight of how to construct a half adder by looking at its truth table on the right.

Co

Inputs		Outputs	
Xo	Yo	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$\underbrace{\&\quad\quad\oplus}$

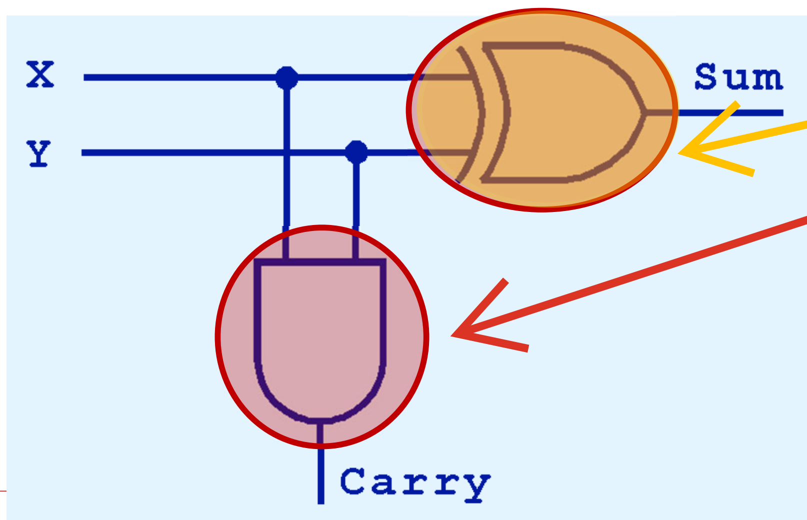
Two 1-bit numbers' addition will result in a 2-bit answer!

Combinational Circuits: *Half Adder* (‘Cont)

- It consists two gates:
  - a XOR gate -- the sum bit
  - a AND gate -- the carry bit

Inputs		Outputs	
Xo	Yo	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

& $\oplus$



Combinational Circuits: *Full (bit) Adder*

two binary numbers
(2-bit system)

	Co	
X =	X1	Xo
Y =	Y1	Yo

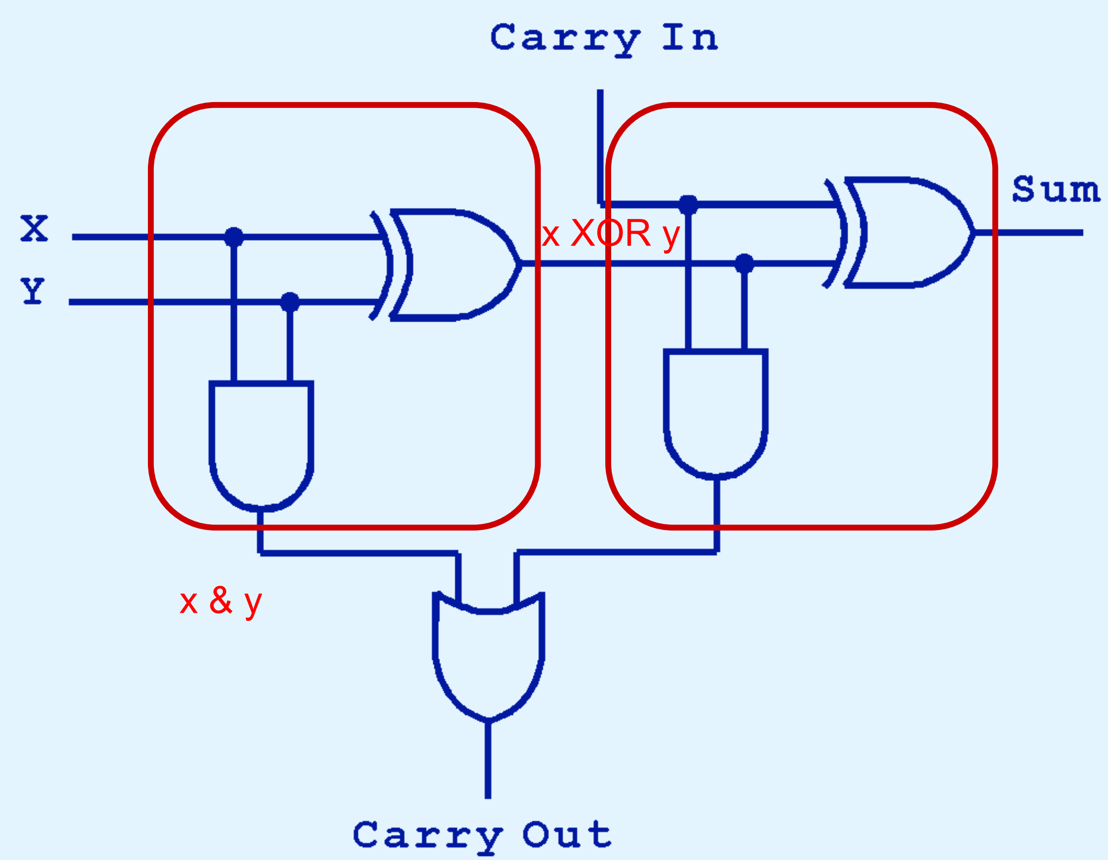
- We can extend the half adder to a full adder, which includes an additional carry bit (Carry In)
- The truth table for a full adder is shown on the right.

Inputs			Outputs	
X1	Y1	Carry In(Co)	Sum	Carry Out
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

Three1-bit numbers' addition will ALSO result in just a 2-bit answer!

The Full Adder (3I-2O circuit)

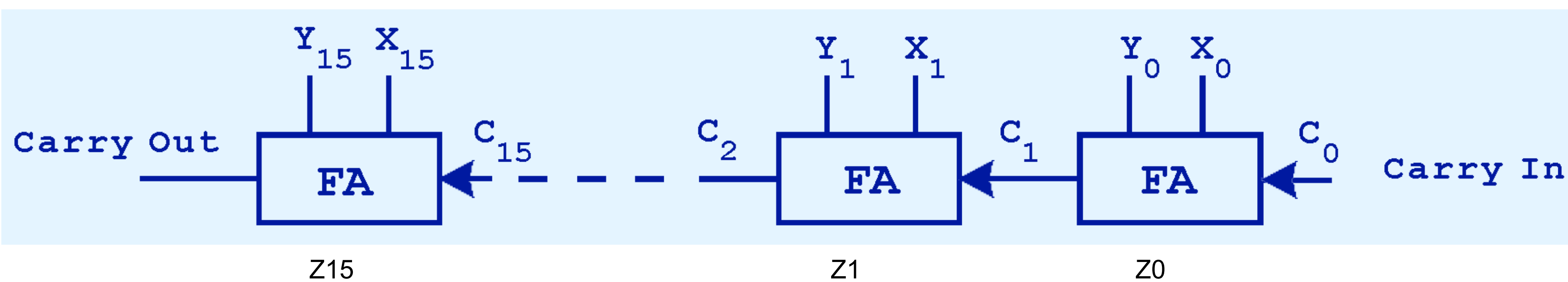
Lower order bits' addition



Inputs			Outputs	
X	Y	Carry In	Sum	Carry Out
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

Ripple-carry Adder

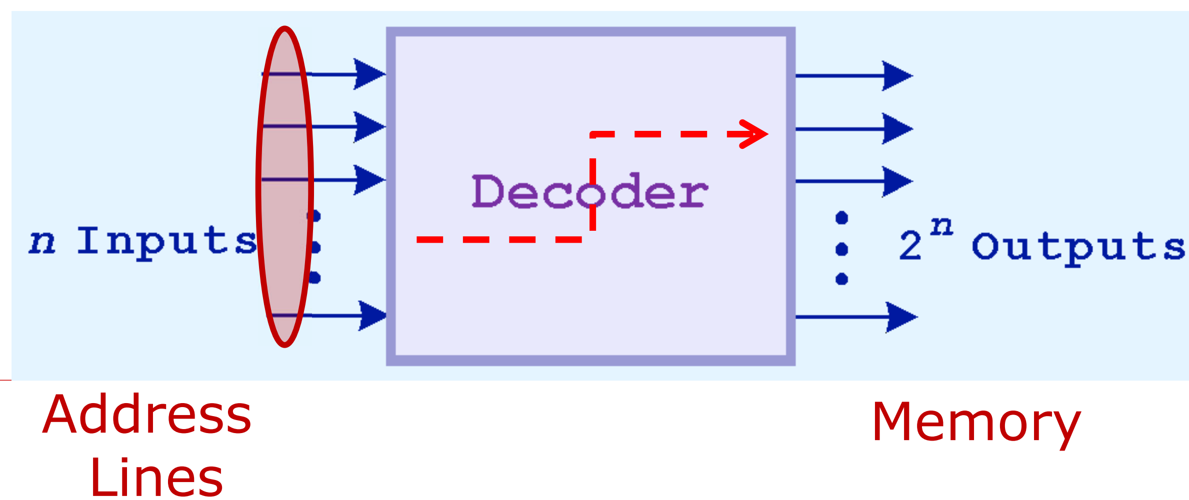
- Just as we combined half adders to construct a full adder, full adders can be connected in series.
- The carry bit "ripples" from one adder to the next. This configuration is called a ***ripple-carry adder***.



- This is the full adder for two 16 bits!

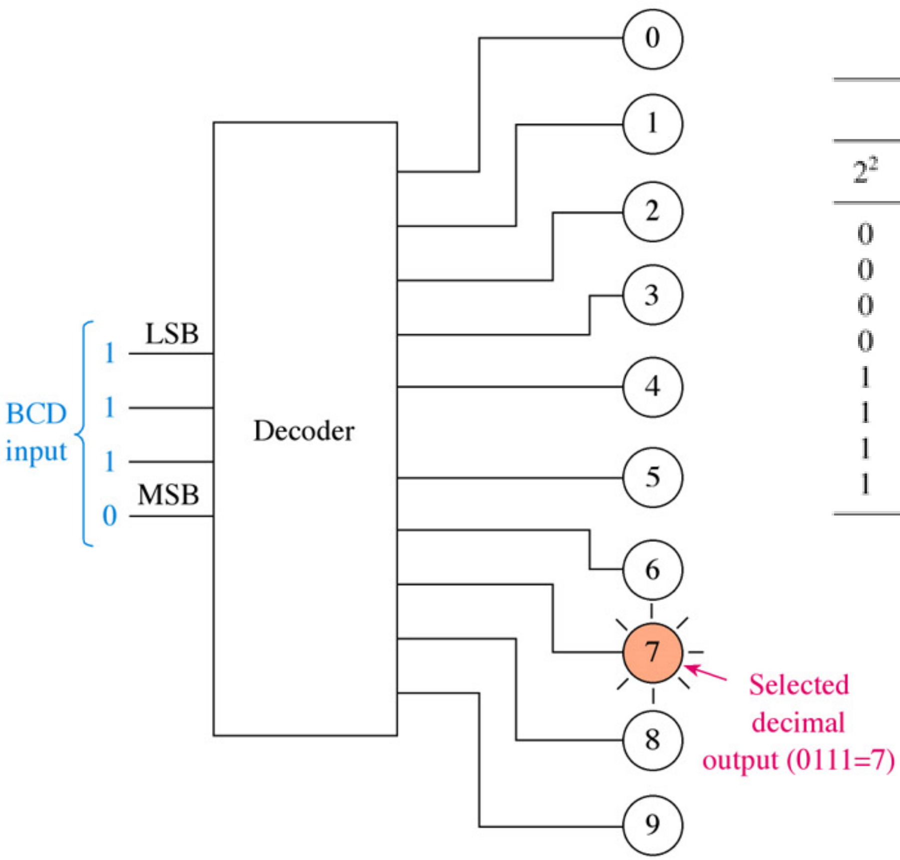
Decoder

- Decoder is another important combinational circuit.
- It is used to select a memory location according a address in binary form
  - Application: given a memory address → Obtain its memory content.
- Address decoder with *n inputs* can select one out of *2^n locations*.



Not 10 but 4 switches to turn 10 bulbs on/off!!

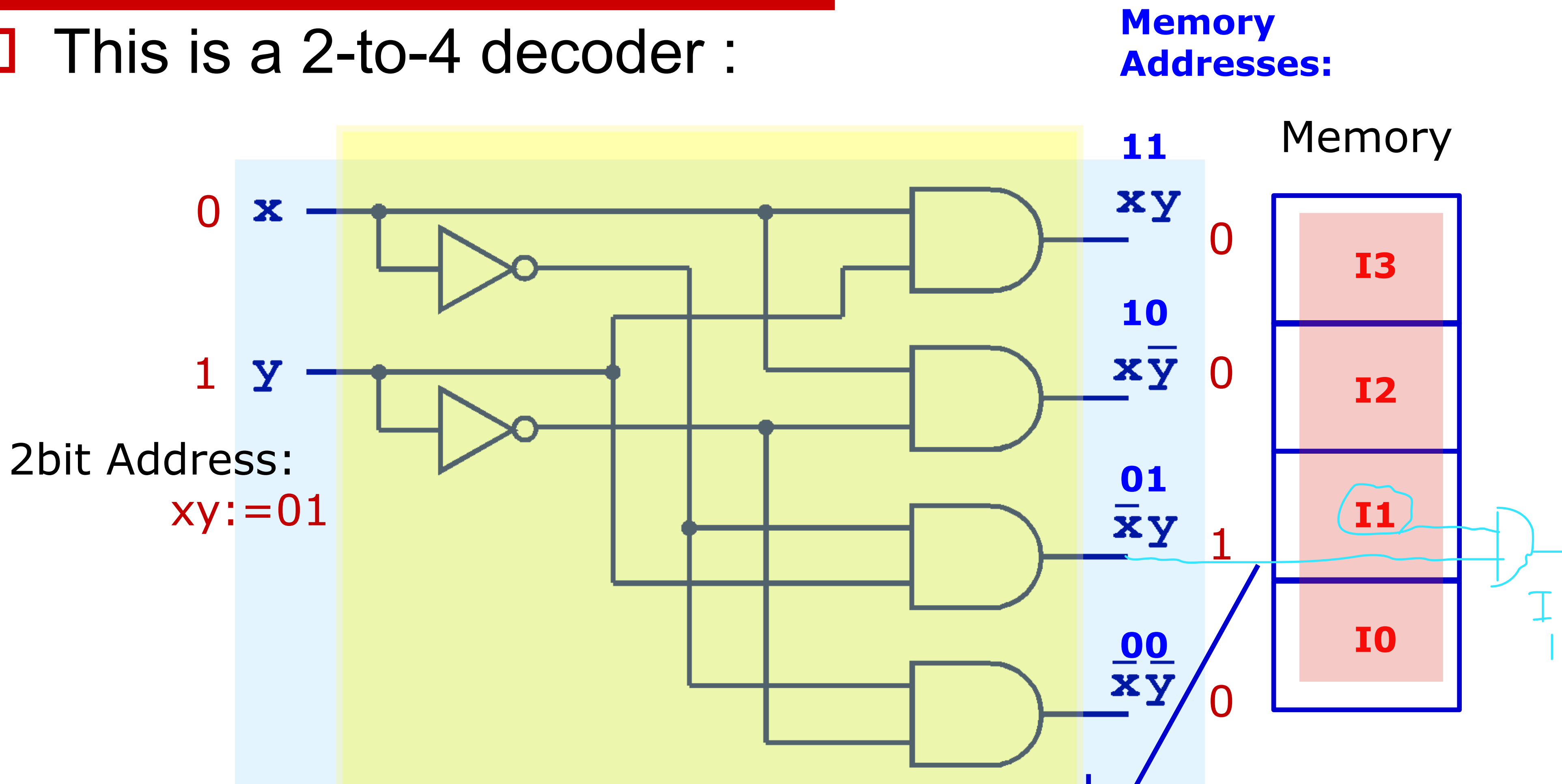
Decoder



Input			Output							
2^2	2^1	2^0	0	1	2	3	4	5	6	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

A 2-to-4 Decoder

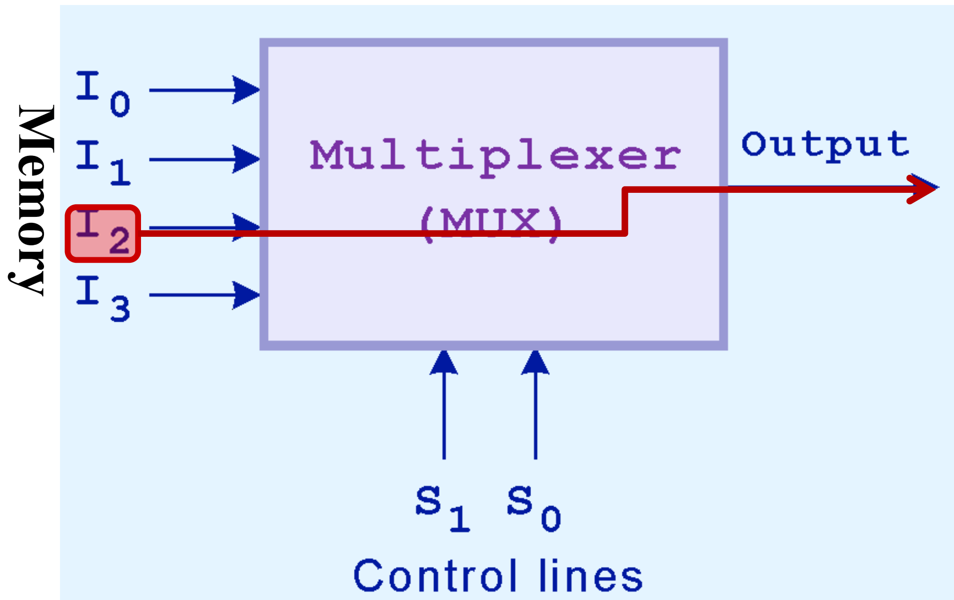
- This is a 2-to-4 decoder :



Only this piece of memory will be chosen/accessed

Multiplexer

- A multiplexer works just the opposite to a decoder.
- It selects a single value from multiple inputs.
- The chosen input for output is determined by the value of the multiplexer's control lines.
- To select from ***n*** inputs, $\log_2 n$ control lines are required.



works like a decoder

1 **0** **Address**

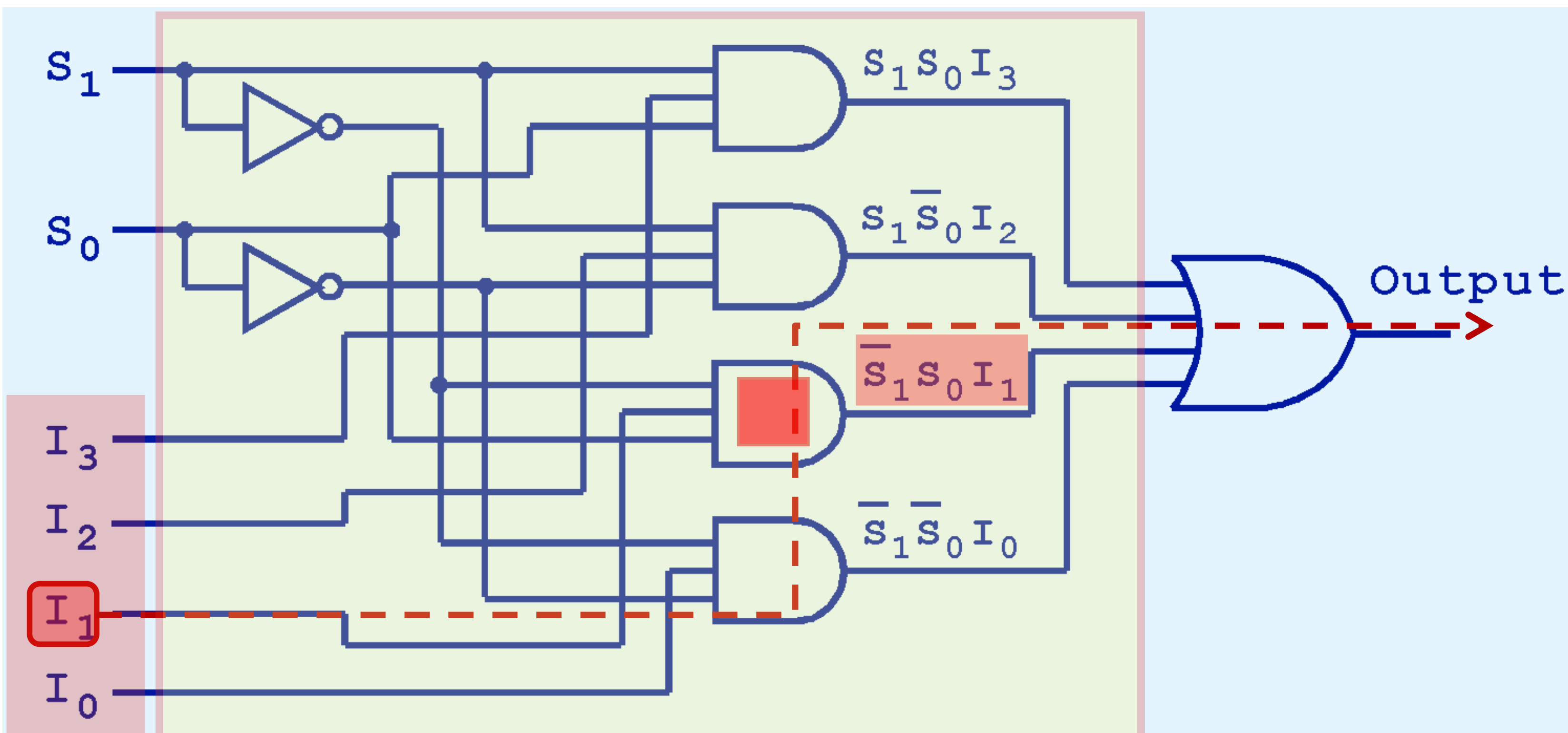
Combinational Circuits

- Using a 2-to-4 decoder, this is a 4-to-1 multiplexer.

0

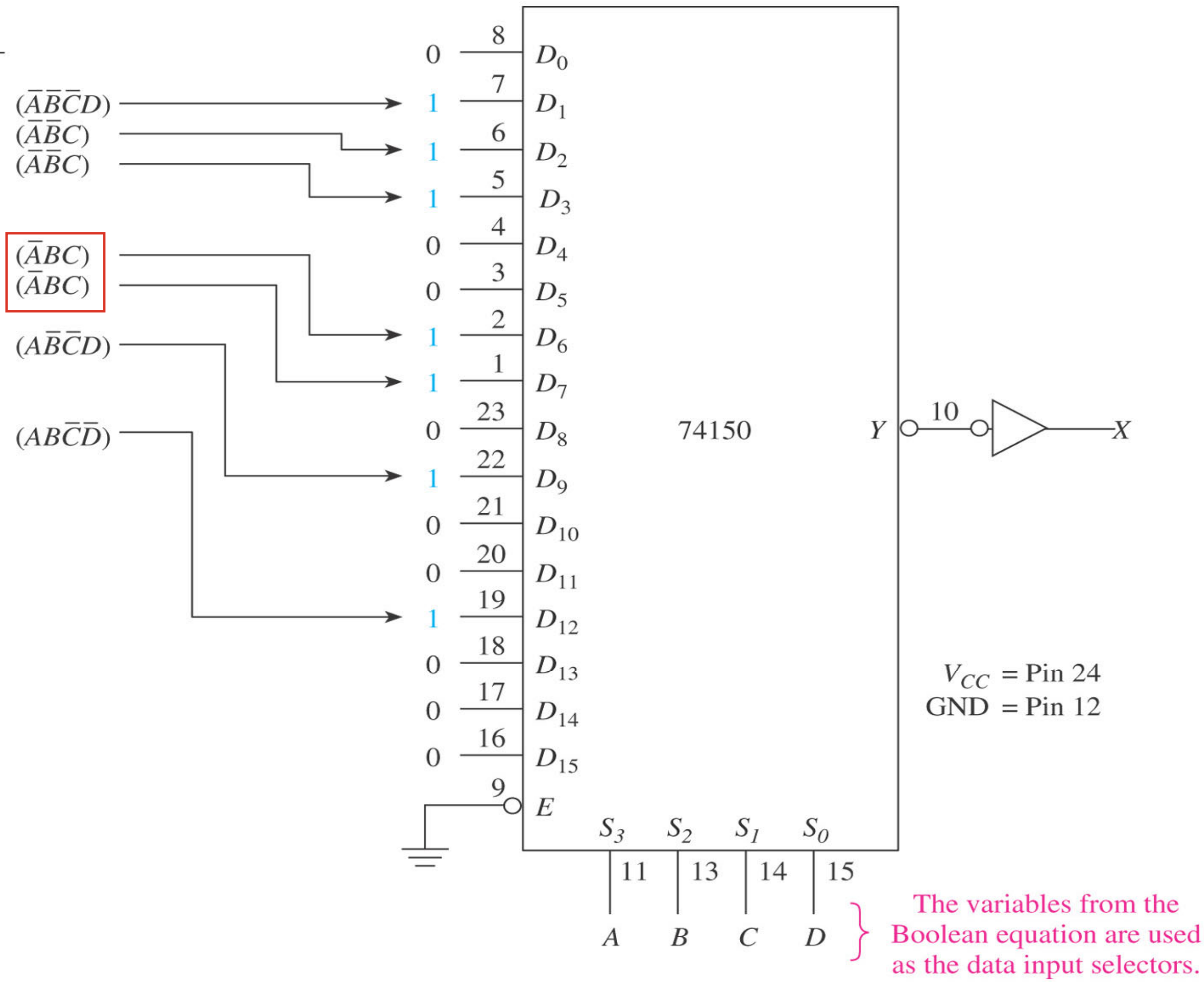
x'y

1



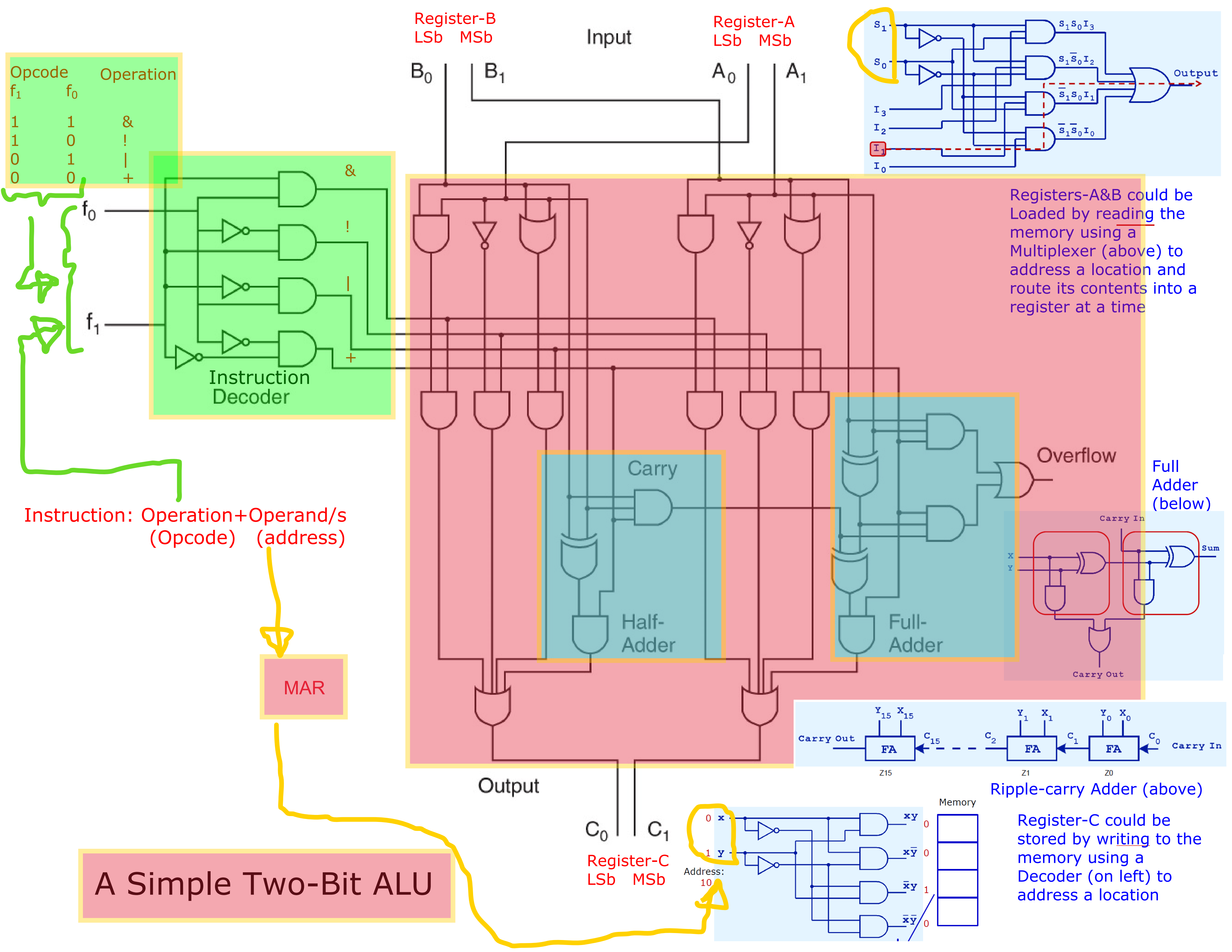
which input is transferred to the output?

A	B	C	D	X
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0



Using a multiplexer to implement the Boolean equation

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



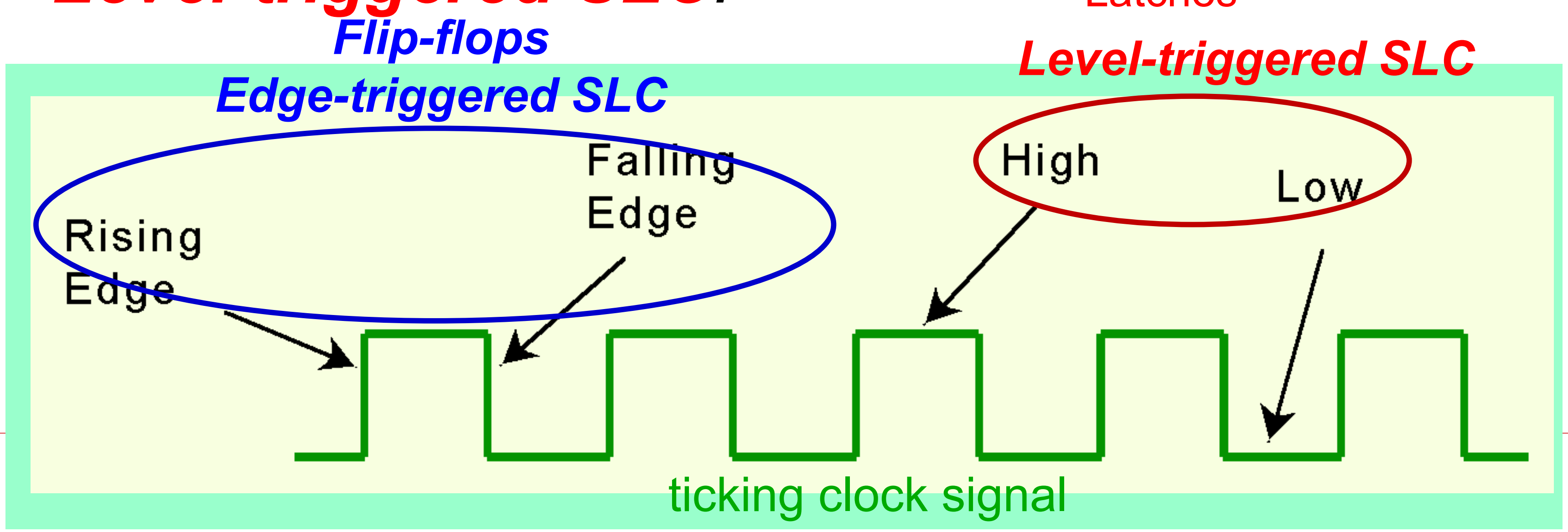
A Simple Two-Bit ALU
Opcode
Operation
f1 f0
1 1 &
1 0 !
0 1 |
0 0 +
f0
f1
Instruction Decoder
Instruction: Operation+Operand/s
(Opcode) (address)
MAR
Register-B
LSb MSb
B0 B1
Input
Register-A
LSb MSb
A0 A1
Carry
Half-Adder
Full-Adder
Overflow
Output
C0 C1
Register-C
LSb MSb
Registers-A&B could be Loaded by reading the memory using a Multiplexer (above) to address a location and route its contents into a register at a time
S1
S0
I3
I2
I1
I0
Output
Full Adder (below)
Carry In
Sum
Carry Out
Carry Out
FA
Carry In
Z15
Z1
Z0
Ripple-carry Adder (above)
Memory
Address:
10
Register-C could be stored by writing to the memory using a Decoder (on left) to address a location

3.6 Sequential Logic Circuits (*SLC*)

- Combinational logic circuits are perfect for those applications when a Boolean function be immediately evaluated, given the current inputs.
  - Examples: multiplexer, ripple-carry adder, shifter, etc
- However, sometimes, we need a kind of circuits that change value by considering the current inputs and its current state.
  - **Memory** is such an example that requires to remember the current state
  - The circuits need to "***remember***" their states.
- *Sequential logic circuits (SLC)* provide this functionality.

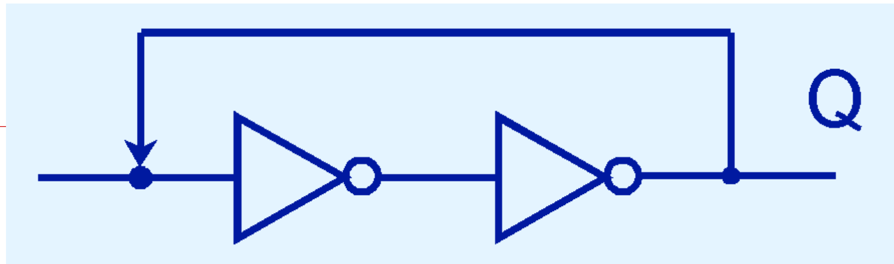
Edge-triggered Or Level-triggered?

- SLC that changes its state at the rising edge, or the falling edge of the clock pulse is called ***Edge-triggered SLC***.
- SLC that changes its state when the clock voltage reaches to its highest or lowest level are called ***Level-triggered SLC***.



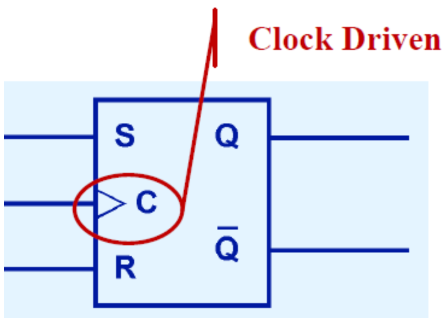
Essential Component Of Sequential Circuits: *Feedback*

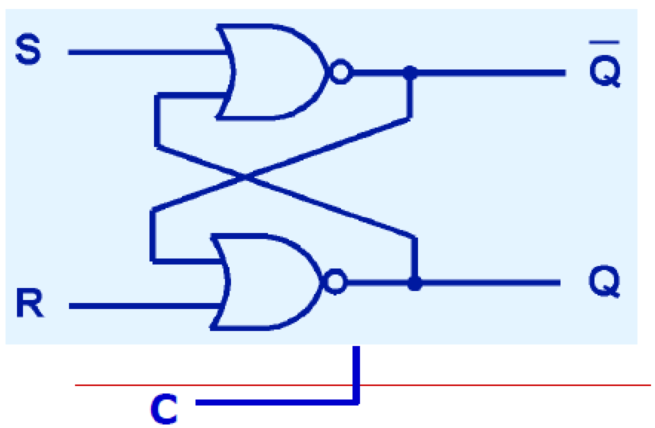
- The most important design mechanism of SLC is ***Feedback***
  - ***Feedback*** can retain the state of sequential circuits
- Feedback in digital circuits occurs when an output is ***looped back*** as an input.
- A simple example of this concept is shown below.
  - If Q is 0 it will always be 0, if it is 1, it will always be 1. --- *The motivation of Memory!*



Behavior Of An SR Flip-flop

- The behavior of an SR flip-flop is illustrated in the following truth table. Note how feedback works.
  - Let's denote Q(t) as the value of the output at time *t*, and
  - Denote Q(t+1) is the value of Q at time *t+1*.





S	R	Q(t+1)
0	0	Q(t) (no change)
0	1	0 (reset to 0)
1	0	1 (set to 1)
1	1	undefined

SR Flip-flop Truth Table

- We consider Q(t), its current output, as the third input for SR flip-flop, besides S and R.
- The truth table for this circuit, as shown on the right.
- When both S and R are 1, the SR flip-flop is in forbidden state

Retain its original value

Change its value

S	R	Present State Q(t)	Next State Q(t+1)
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	undefined
1	1	1	undefined

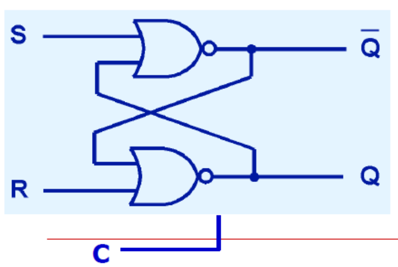
Q(t+1) =Q(t)

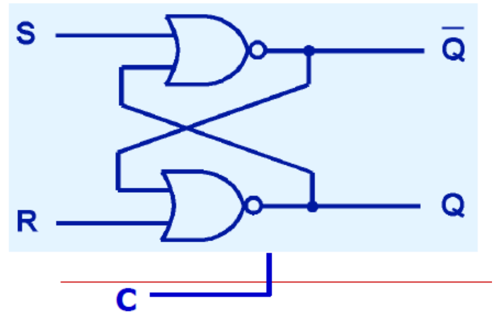
0

1

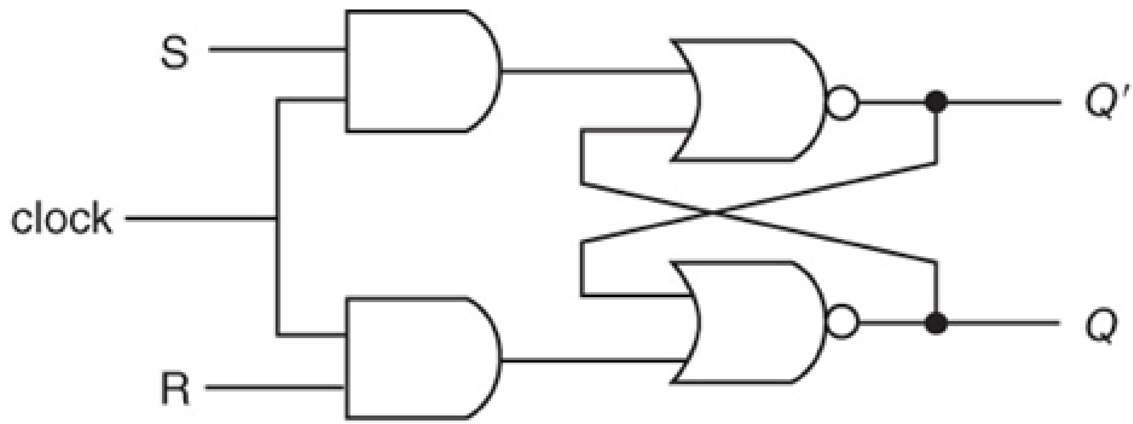
Q=Q'=0

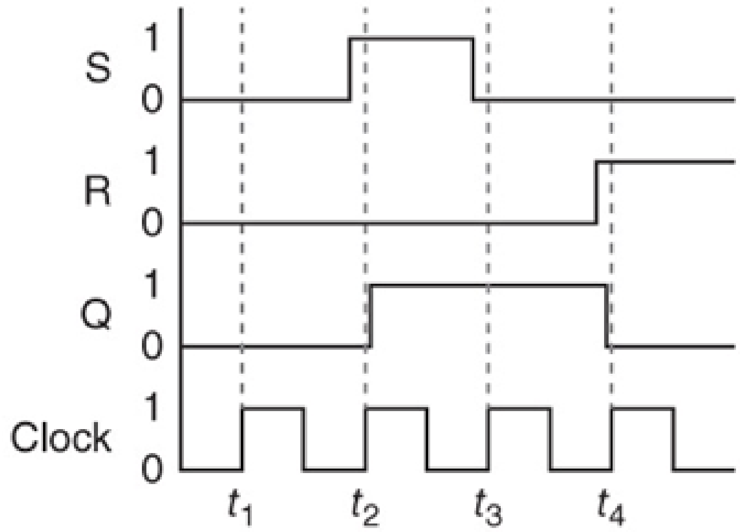
forbidden state





Clocked SR Flip-flop



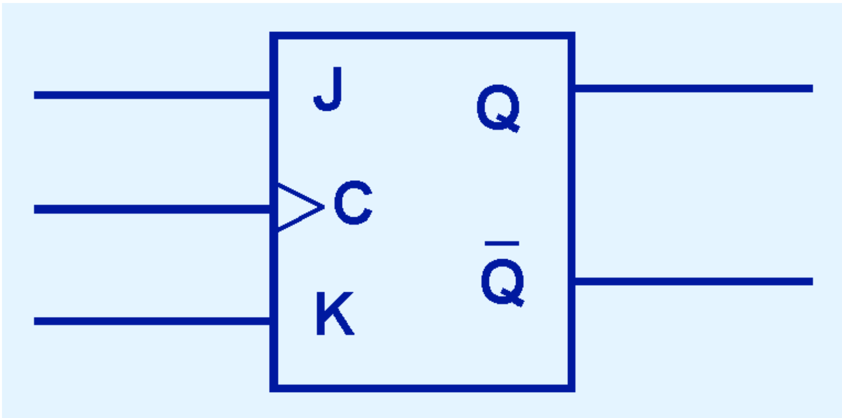


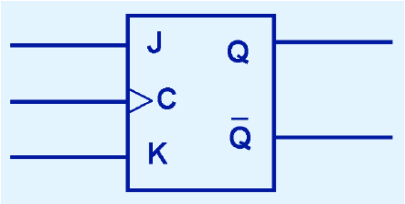
clock (rising edge) enables S or R!

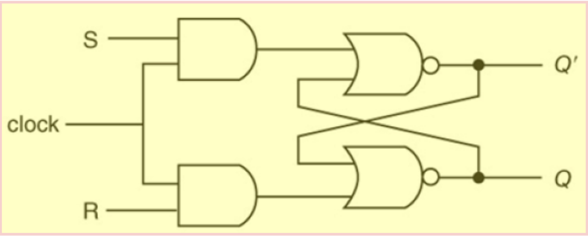
JK Flip-flop

- One limitation of SR flip-flop is that, when S and R are both 1, the output is ***undefined***.
  - This is not nice because it wastes a state
- Therefore, SR flip-flop can be modified to provide a stable state when both S and R inputs are 1.

- This modified flip-flop is called a JK flip-flop, shown on the right.
  - The "JK" is in honor of Jack Kilby.

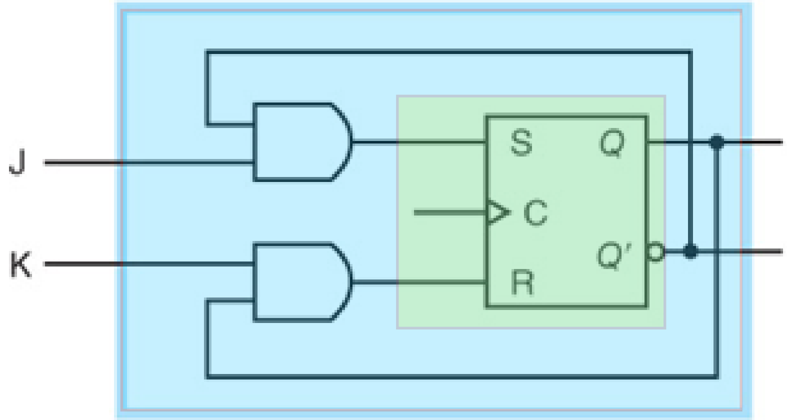






3.6 Sequential Circuits

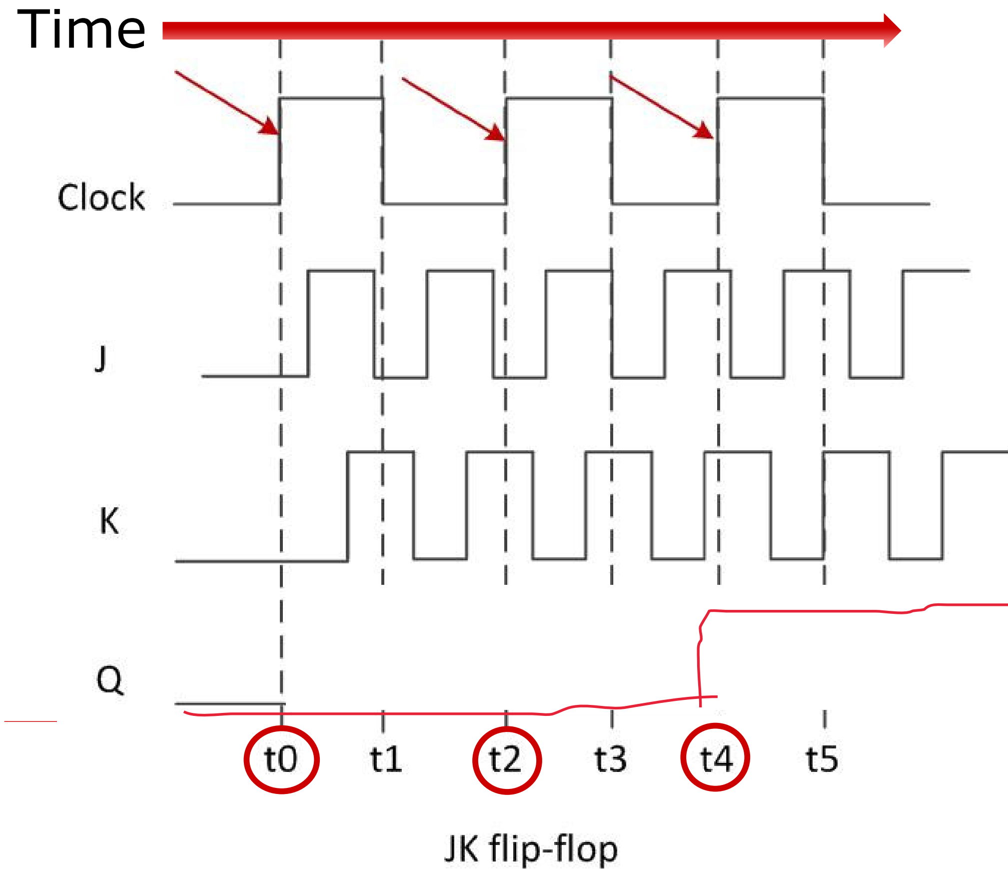
- On the right, we see how an SR flip-flop can be modified to create a JK flip-flop.
- The truth table indicates that the flip-flop is stable for all inputs.
  - When J and K are both 1, Q(t+1) = ¬Q(t)



J	K	Q(t+1)
0	0	Q(t) (no change)
0	1	0 (reset to 0)
1	0	1 (set to 1)
1	1	$\overline{Q}$(t)

An Example

- Let's say a JK flip-flop is ***rising-edge*** triggered
- At t_0, $Q(t) = 0$. What will be the changes of the value of Q over time?

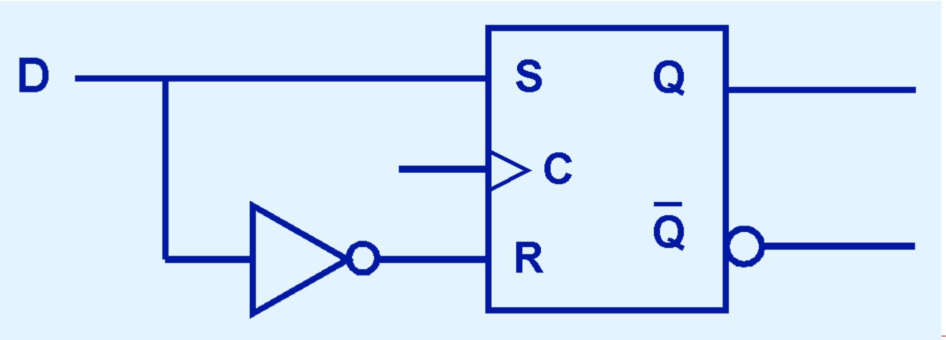


- Any time other than the rising edge **won't** trigger this JK flip-flop to change its state

J	K	Q(t+1)
0	0	Q(t) (no change)
0	1	0 (reset to 0)
1	0	1 (set to 1)
1	1	$\bar{Q}(t)$

D Flip-flop

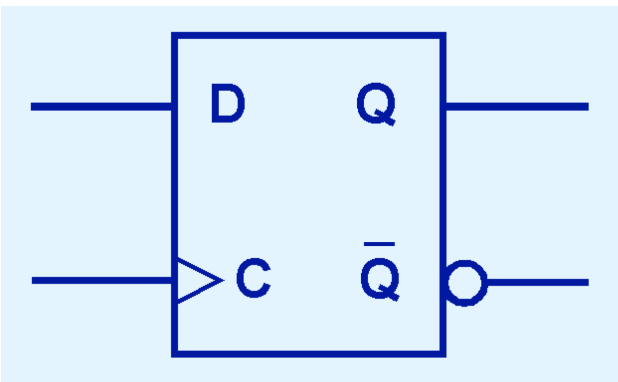
- Another modification of the SR flip-flop is the D flip-flop, shown below with its truth table.
- You will notice that the output of the flip-flop remains the same during subsequent clock pulses. The output changes only when the value of D changes.



D	Q(t+1)
0	0
1	1

D Flip-flop = 1-bit memory

- The D flip-flop is the fundamental circuit of computer **memory**.
  - D flip-flop and its truth table are illustrated as below.

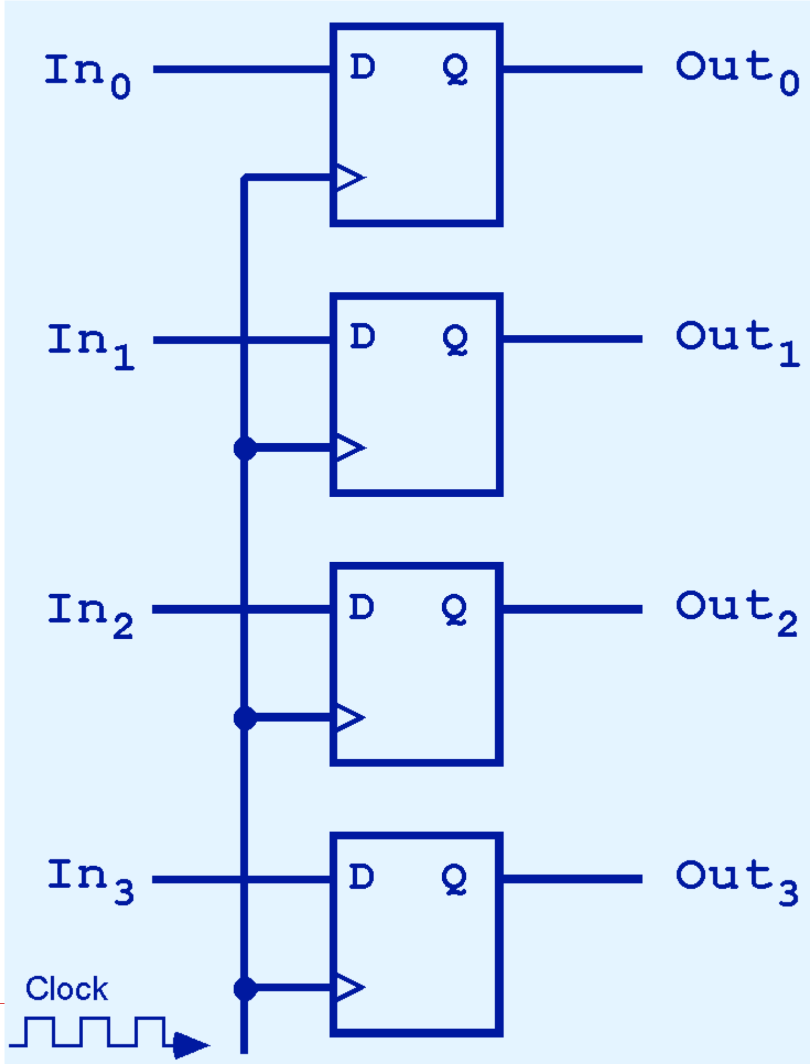


D	Q(t+1)
0	0
1	1

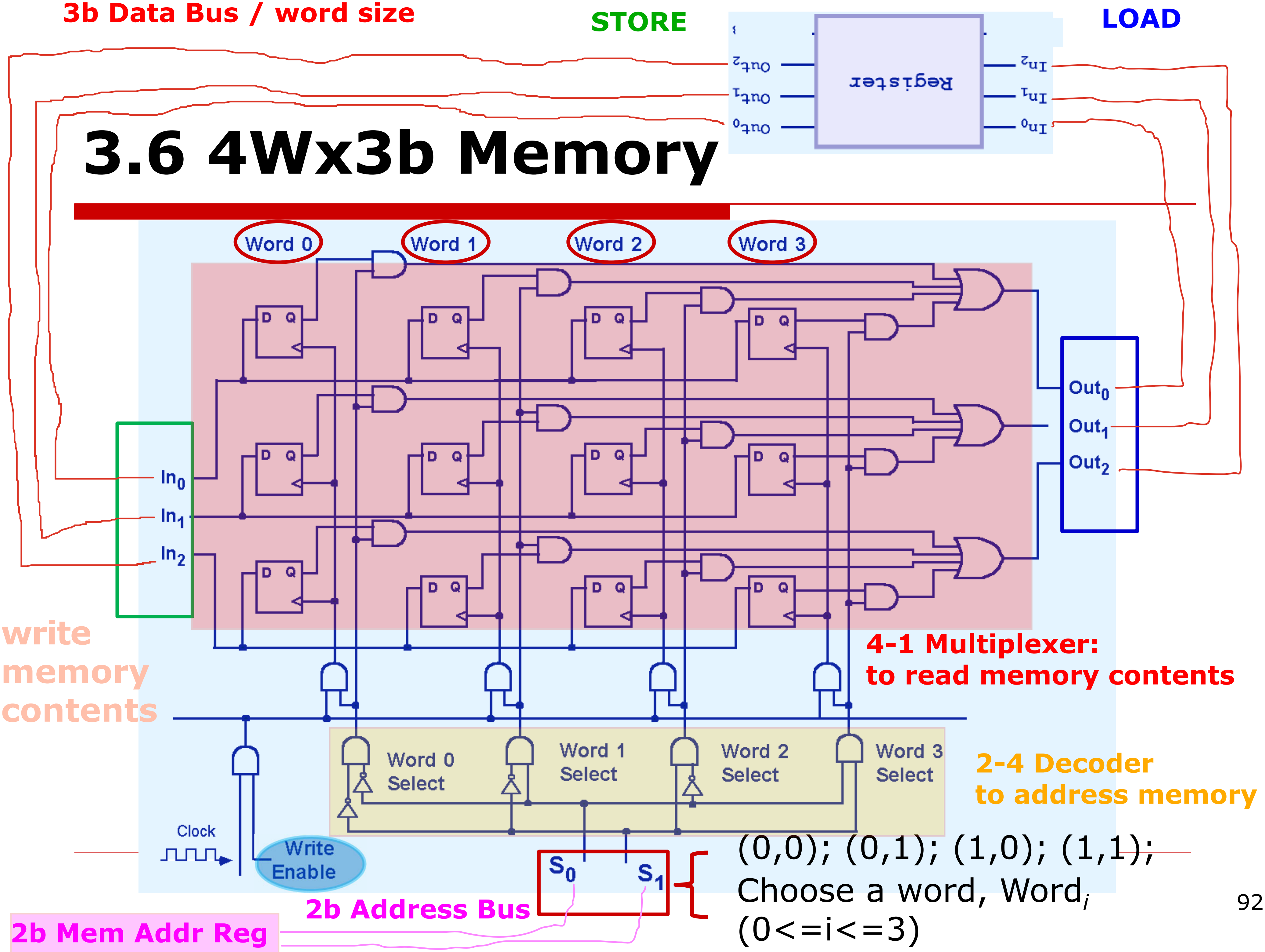
3.6 Sequential Circuits: Register

- This illustration shows a 4-bit register consisting of D flip-flops. You will usually see its block diagram (below) instead.

A larger memory configuration is shown on the next slide.



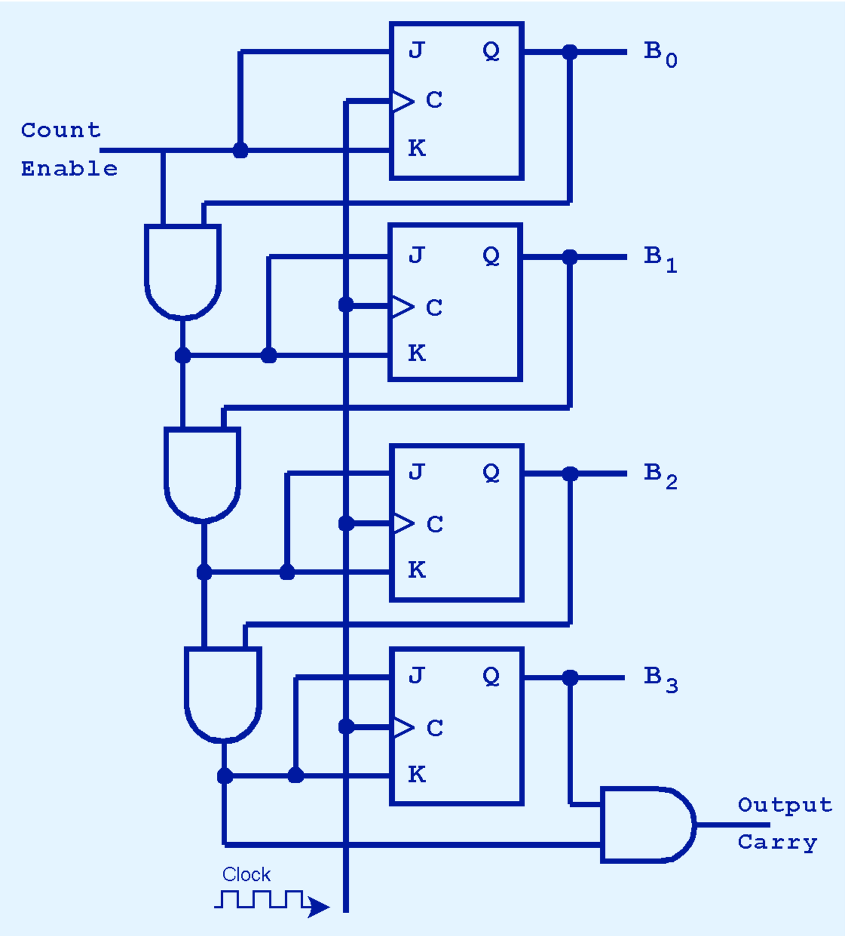
3.6 4Wx3b Memory



(0,0); (0,1); (1,0); (1,1);
Choose a word, $Word_i$
(0<=i<=3)

3.6 Sequential Circuits: Counter

- A binary **counter** is another example of a sequential circuit.
- The low-order bit is **complemented** at each clock pulse.
- Whenever it changes from 0 to 1, the next bit is complemented, and so on through the other flip-flops.



Synchronous MOD-16 counter

3.6 Sequential Circuits: State Machines

$Q(t+1) = f\{Q(t), S, R\}$

SM = CLC+SLC

- Sequential circuits are used anytime that we need to design a "**stateful**" application.
  - A stateful application is one where the next state of the machine depends on the current state of the machine and the input.
- A stateful application **requires both combinational and sequential logic**.
- The following slides provide several examples of circuits that fall into this category.

Can you think of others?

3.7 Designing Circuits

- Digital designers rely on **specialized software to create efficient circuits**.
  - Thus, software is an enabler for the construction of better hardware.
- Of course, **software** is in reality a collection of algorithms that could just as well be **implemented in hardware**.
  - Recall the **Principle of Equivalence of Hardware and Software**.

Designing Circuits

- When we need to implement a **simple, specialized algorithm** and its execution **speed** must be as **fast** as possible, a **hardware solution is often preferred**.
- This is the idea behind ***embedded systems***, which are small special-purpose computers that we find in many everyday things.
- Embedded systems **require special programming** that demands an **understanding of the operation of digital circuits**, the basics of which you have learned in this chapter.

- Assembly programming for performance. FPGA (hardware) design using HDL (e.g. VHDL, Verilog) languages!

Chapter 3 Conclusion

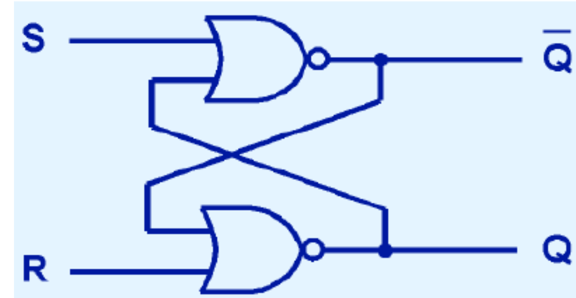
- Computers are implementations of Boolean logic.
- Boolean functions are completely described by truth tables.
- Logic gates are small circuits that implement Boolean operators.
- The basic gates are AND, OR, and NOT.
  - The XOR gate is very useful in parity checkers and adders.
- The “universal gates” are NOR and NAND.

Chapter 3 Conclusion

- **Computer circuits** consist of **combinational** logic circuits and **sequential** logic circuits.
- **Combinational** circuits **produce outputs almost immediately** when their inputs change.
- **Sequential** circuits **require clocks** to control their changes of state.
- The basic sequential circuit unit is the **flip-flop**: The behaviors of the **SR, JK, and D** flip-flops are the most important to know.

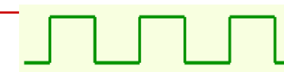
0) CLC >>>>

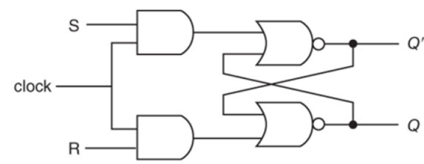
1) got on/off controls.

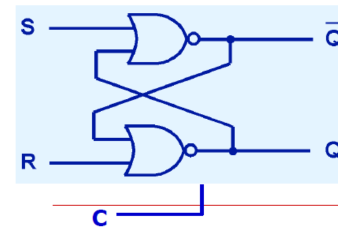


End of Chapter 3 Summary

2) clock trigger



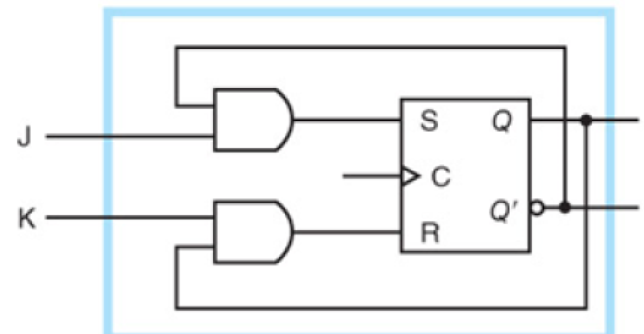




2b) mostly defined synchronized states.

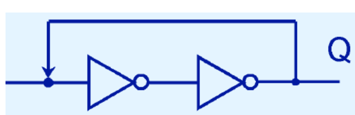
3) fully defined states

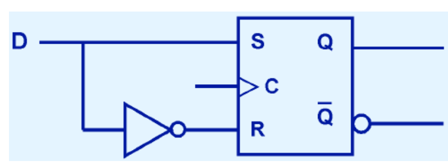
"error free"!

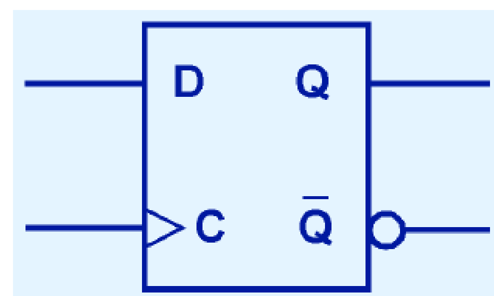


4) Memory

"Preserving state"







Chapter 3 Summary

- Simplify using Boolean algebra and canonical forms
- Digital logic circuits: simplifications, apps (error detector, half/full/RC adder, decoder, multiplexer, ALU)
- SLC: SR-FF, JK-FF, D-FF, Register/Memory cells, Counter, State Machines,