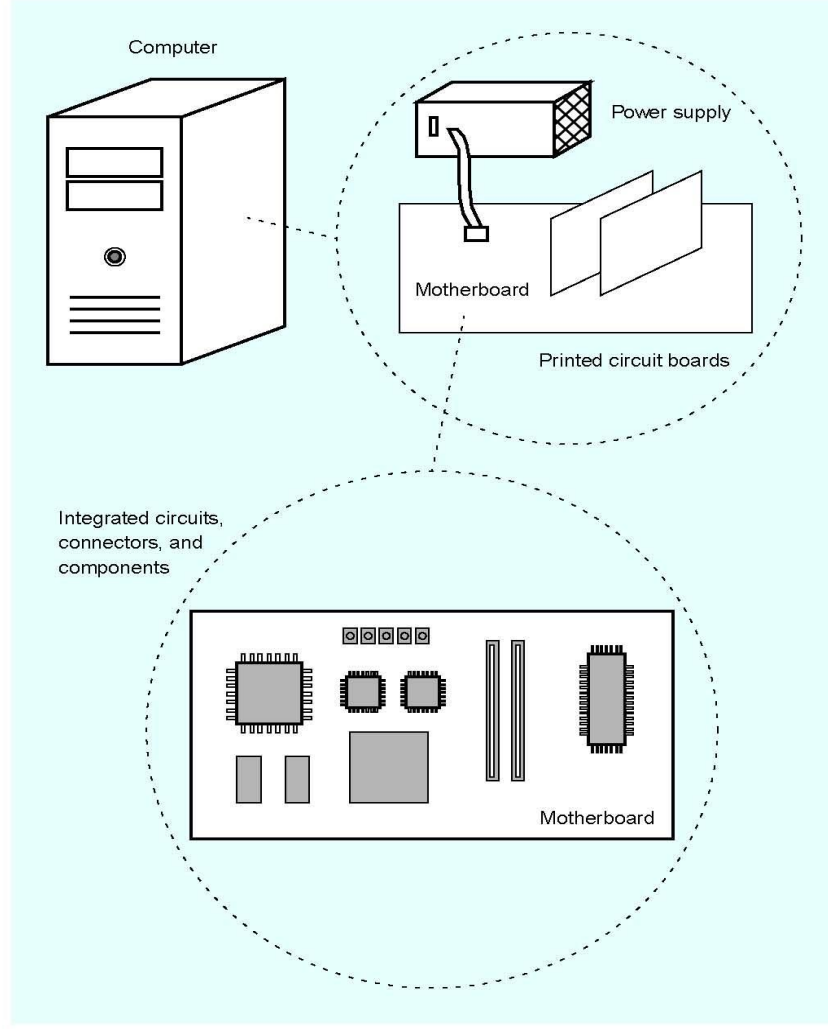
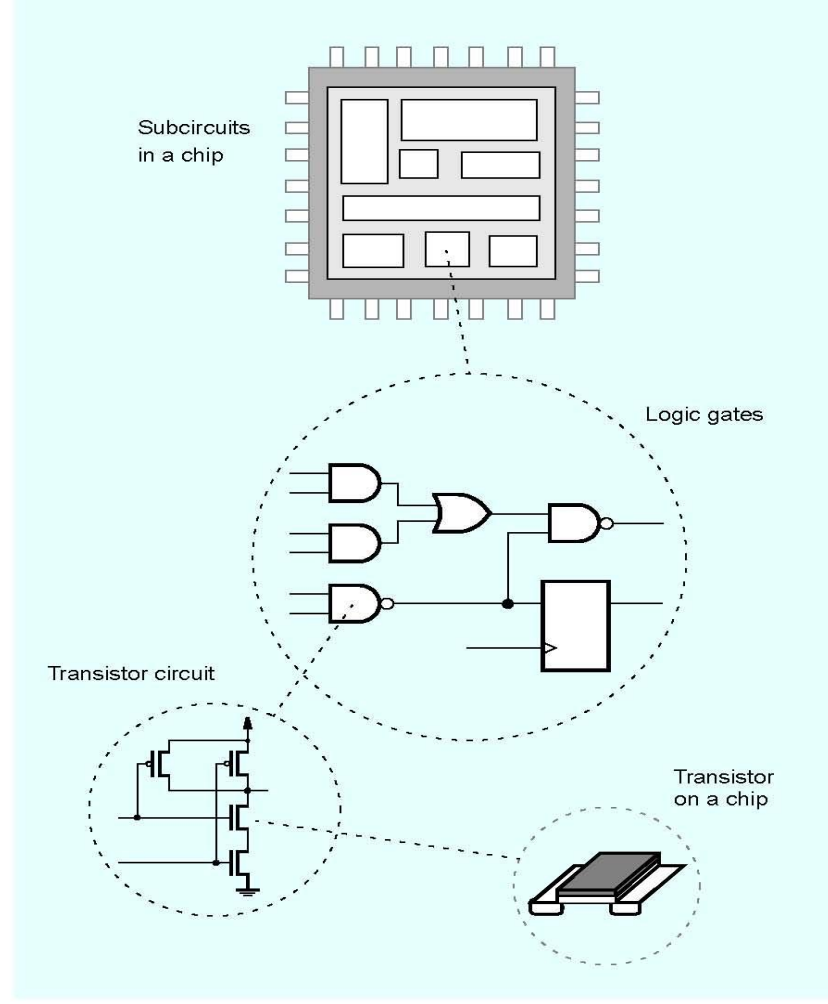
Digital System Hierarchy

To understand the role that logic circuits play in digital systems, let's look at the structure of a typical computer.



Digital System Hierarchy

As shown in the middle of the figure, a logic circuit comprises a network of connected logic gates.



Digital Hardware

Digital hardware products are often built from basic logic circuits implemented in integrated circuits (ICs). Here is a range of ICs that are commonly used:

- Standard Chips – Discrete implementation of common logic functions
- Programmable Logic Devices (PLDs) – chips that contain circuitry that can be configured by the user to implement a wide range of different logic circuits.
- Field Programmable Gate Arrays (FPGAs) – More complex than PLDs. Still reconfigurable hardware
- Masked technology: Application Specific ICs (ASICs) - (not reconfigurable)
  - Types include Gate Array, Standard Cell, Full Custom

Computer Aided Design

- Logic circuits can be represented a number of different ways. The previous example showed a schematic representation.

- Modern design of logic circuits depends heavily on Computer Aided Design (CAD) software tools. These tools simplify the design and testing of a logic circuit.

- In this course we will use Verilog to define logic designs.

Other Terms or Concepts

- Boolean Logic: a complete system for logical operations, used in many disciplines. It was named after George Boole, who first defined an algebraic system of logic in the mid 19th century. (Ref: Wikipedia)

- In terms of complexity, logic circuits consist of:
  - Combinational logic – Outputs are a function of only the inputs. These are implemented using Logic Gates.
  - Sequential logic – Outputs are a function of both the inputs and current outputs. These are implemented using Flip-Flops and Registers. (Both of these components have memory.)

Variables and Functions

Boolean variables – can take one of the two values 0 (false) and 1 (true).

The simplest binary element is a *switch* that has two states. If a given switch is controlled by an input variable x, then we will say that the switch is open if x=0 and closed if x=1.

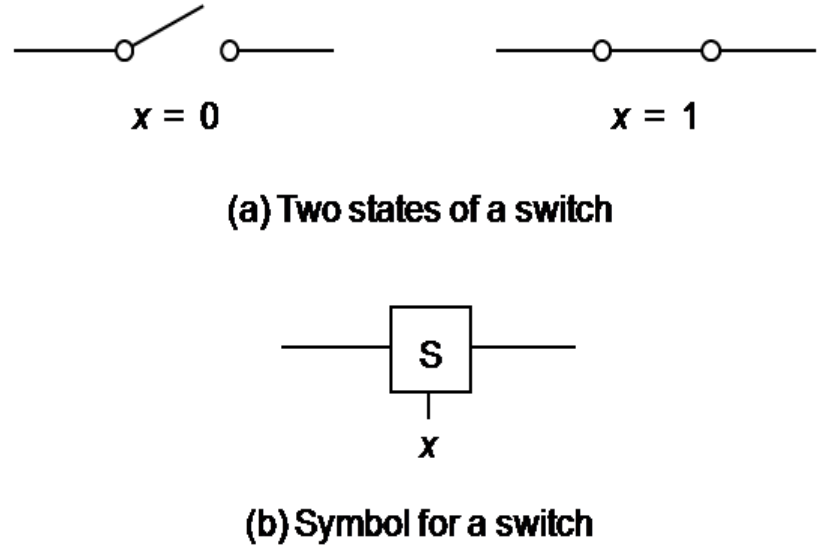


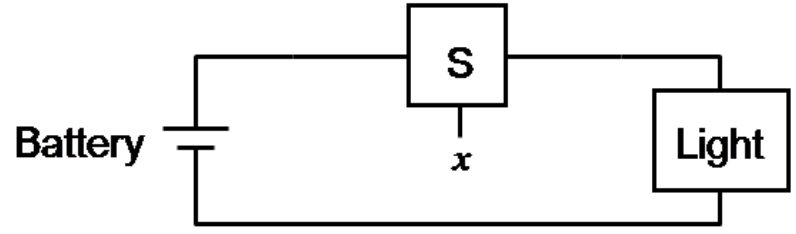
Figure 2.1. A binary switch.

Variables and Functions - continued

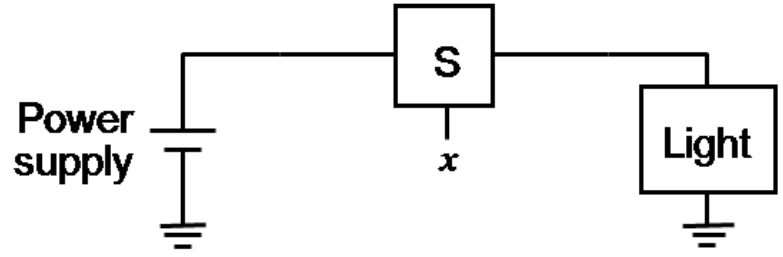
The switch turns a light bulb on or off.

If the output (denoted as L) is defined as the state of the light, we say that L=0 when the light is off; L=1 when the light is on. We can describe the state of the light as a function of the input variable x.

$$L(x) = x$$



(a) Simple connection to a battery



(b) Using a ground connection as the return path

Figure 2.2. A light controlled by a switch.

Variables and Functions - continued

logic AND function:

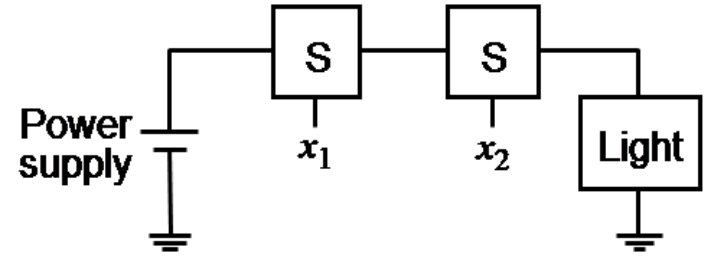
$$L(x_1, x_2) = x_1 \cdot x_2$$

where L=1 if x_1=1 and x_2=1; L=0 otherwise. The "·" is the ***AND operator***.

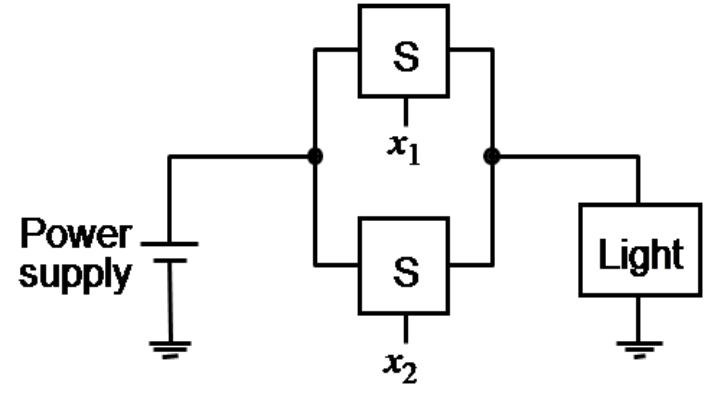
logic OR function:

$$L(x_1, x_2) = x_1 + x_2$$

where L=1 if x_1=1 or x_2=1 or if x_1=x_2=1 ; L=0 if x_1=x_2=0. The "+" is the ***OR operator***.



(a) The logical AND function (series connection)



(b) The logical OR function (parallel connection)

Figure 2.3. Two basic functions.

Inversion or Complement

$$L(x) = \overline{x}$$

where L=1 if x=0; L=0 if x=1.

NOT operators: ¯ ′ ! ~

$$\overline{x} = x' = !x = \sim x$$

The NOT operator can also be applied to complex expressions

$$\overline{f}(x_1, x_2) = \overline{x_1 + x_2}$$

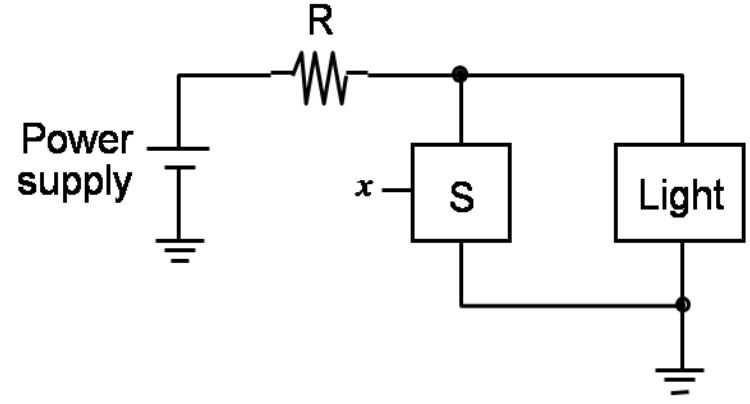


Figure 2.5. An inverting circuit.

Truth Table

Logic operations (e.g. *AND*, *OR*) can also be represented by Truth Table that assigns an output value for each combination of input values.

Truth tables can also be used to depict information involving complex logic functions.

Concept of Binary counting

x_1	x_2	$x_1 \cdot x_2$	$x_1 + x_2$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1
		AND	OR

Figure 2.6. A truth table for the AND and OR operations.

Truth Table - continued

Truth tables grow exponentially in size with the number input variables.

In general, for n input variables the truth table has 2^n rows.

x_1	x_2	x_3	$x_1 \cdot x_2 \cdot x_3$	$x_1 + x_2 + x_3$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 2.7. Three-input AND and OR operations.

Logic Gates and Networks

The three basic logic operations (AND, OR, NOT) can be used to implement logic functions of any complexity.

Each logic operation can be implemented electronically with transistors, resulting in a circuit element called a *logic gate*.

We use graphical symbols to represent logic gates.

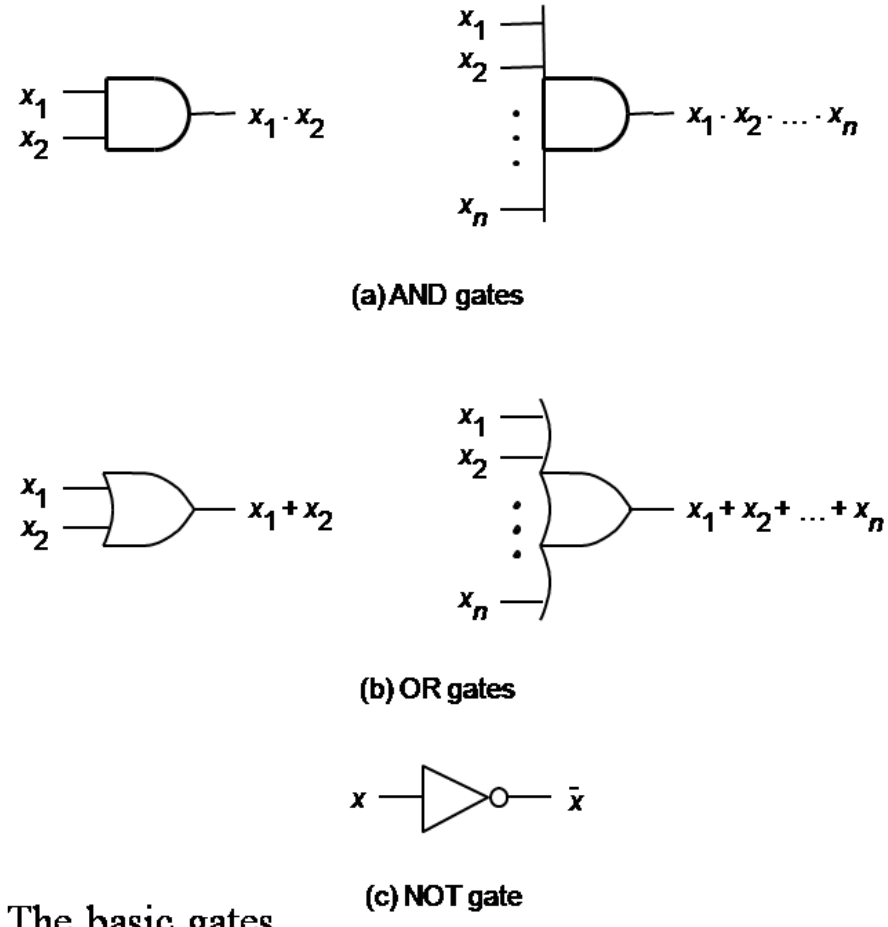
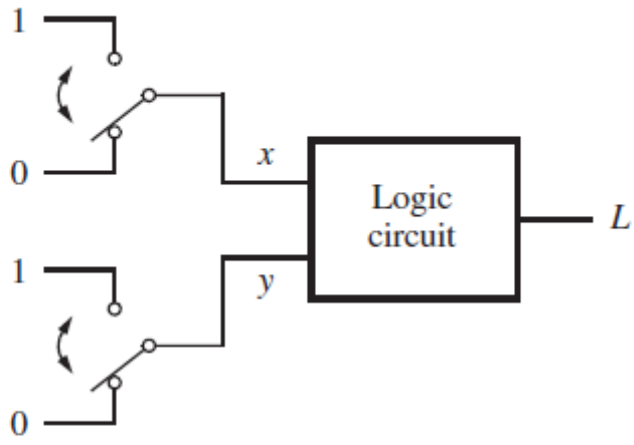


Figure 2.8. The basic gates.

Digital Representation of Information

- 8 bits (8 0s or 1s) can represent 2^8 = 256 different patterns.
- These 256 patterns can be assigned to anything that serves a purpose.
  - Unsigned integers (0 to 255)
  - Signed integers (-128 to 127)
  - Fixed-point numbers (0, 0.25, 0.5, 0.75, 1.0 … 63.0, 63.25, 63.75)
  - Two Binary Coded Decimal (BCD) digits (0111 0011 = 73)
  - A character of text (ASCII or plain text: 96 printable, 32 control, 128 other)
  - The results of flipping a coin 8 times (0 = heads, 1 = tails)
- 64 bits can represent 2^64 =~ 1.845*10^19 different patterns.
  - Floating point numbers (-10^308 to -10^-308, 0, 10^-308 to 10^-308, to 15 significant digits.
- Bits can represent numbers, text, audio, images, video, 3D objects, etc.
- Managing bits allows managing all these real-world objects.

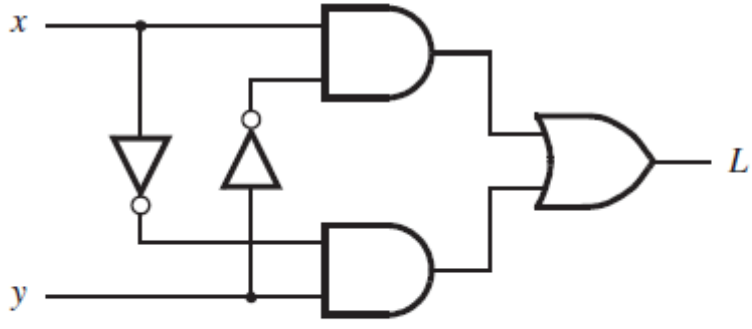
Logic Network- example



(a) Two switches that control a light

x	y	L
0	0	0
0	1	1
1	0	1
1	1	0

(b) Truth table



(c) Logic network

(d) XOR gate symbol

Figure 2.11. An example of a logic circuit.

Boolean Algebra

- Boolean expressions – contain boolean variables (or constants) and basic logic operators. E.g.

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

Boolean Algebra provides a powerful tool that can be used for designing and analyzing logic circuits.

Boolean Algebra – Axioms & Theorems

- Axioms – basic assumptions

$1a.\quad 0 \cdot 0 = 0$	$1b.\quad 1 + 1 = 1$
$2a.\quad 1 \cdot 1 = 1$	$2b.\quad 0 + 0 = 0$
$3a.\quad 0 \cdot 1 = 1 \cdot 0 = 0$	$3b.\quad 1 + 0 = 0 + 1 = 1$
$4a.\quad if\ x = 0, then\ \bar{x} = 1$	$4b.\quad if\ x = 1, then\ \bar{x} = 0$

- Basic Theorems – single variable

$5a.\quad x \cdot 0 = 0$	$5b.\quad x + 1 = 1$
$6a.\quad x \cdot 1 = x$	$6b.\quad x + 0 = x$
$7a.\quad x \cdot x = x$	$7b.\quad x + x = x$
$8a.\quad x \cdot \bar{x} = 0$	$8b.\quad x + \bar{x} = 1$
$9.\quad \bar{\bar{x}} = x$	

Boolean Algebra – More Properties

10*a*.	$x \cdot y = y \cdot x$	*Commutative*
10*b*.	$x + y = y + x$	
11*a*.	$x \cdot (y \cdot z) = (x \cdot y) \cdot z$	*Associative*
11*b*.	$x + (y + z) = (x + y) + z$	
12*a*.	$x \cdot (y + z) = x \cdot y + x \cdot z$	*Distributive*
12*b*.	$x + (y \cdot z) = (x + y) \cdot (x + z)$	
13*a*.	$x + x \cdot y = x$	*Absoption*
13*b*.	$x \cdot (x + y) = x$	

Boolean Algebra – More Properties

14*a*.	$x \cdot y + x \cdot \overline{y} = x$	*Combining*
14*b*.	$(x + y) \cdot (x + \overline{y}) = x$	
15*a*.	$\overline{x \cdot y} = \overline{x} + \overline{y}$	*DeMorgan's theorem*
15*b*.	$\overline{x + y} = \overline{x} \cdot \overline{y}$	
16*a*.	$x + \overline{x} \cdot y = x + y$	
16*b*.	$x \cdot (\overline{x} + y) = x \cdot y$	
17*a*.	$x \cdot y + y \cdot z + \overline{x} \cdot z = x \cdot y + \overline{x} \cdot z$	*Consensus*
17*b*.	$(x + y) \cdot (y + z) \cdot (\overline{x} + z) = (x + y) \cdot (\overline{x} + z)$	

Boolean Algebra – algebraic manipulation

The preceding axioms, theorems, and properties provide information necessary for performing algebraic manipulation of more complex expressions. For example, let us prove the validity of the logic equation:

$$x_1 \bullet \overline{x_3} + \overline{x_2} \bullet \overline{x_3} + x_1 \bullet x_3 + \overline{x_2} \bullet x_3 = \overline{x_1} \bullet \overline{x_2} + x_1 \bullet x_2 + x_1 \bullet \overline{x_2}$$

$$\begin{aligned} LHS &= x_1 \bullet \overline{x_3} + x_1 \bullet x_3 + \overline{x_2} \bullet \overline{x_3} + \overline{x_2} \bullet x_3 && 10b \\ &= x_1 \bullet \left(\overline{x_3} + x_3\right) + \overline{x_2} \bullet \left(\overline{x_3} + x_3\right) && 12a \\ &= x_1 \bullet 1 + \overline{x_2} \bullet 1 && 8b \\ &= x_1 + \overline{x_2} && 6a \end{aligned}$$

$$\begin{aligned} RHS &= \overline{x_1} \bullet \overline{x_2} + x_1 \bullet \left(x_2 + \overline{x_2}\right) && 12a \\ &= \overline{x_1} \bullet \overline{x_2} + x_1 \bullet 1 && 8b \\ &= \overline{x_1} \bullet \overline{x_2} + x_1 && 6a \\ &= x_1 + \overline{x_1} \bullet \overline{x_2} && 10b \\ &= x_1 + \overline{x_2} && 16a \end{aligned}$$

Ways of describing Logical Functions

- **Textual Description (“Requirements” or “Specification”)**
- **Truth Table**
- **Minterms (∑m(…)) or Maxterms (∏(M(…))**
- **Logic expression (standard SOP or POS, simplified)**
- **Timing Diagram**
- **Venn Diagram**
- **Switches**
- **Logic circuit (schematic, gates)**
- **Hardware description language (Verilog)**

Hardware Description Languages

- A Hardware Description Language (HDL) is similar to a computer programming language (C, C++, Java, Python…) except that it describes hardware.
- Common HDLs
  - VHDL (Very high speed integrated circuit HDL)
  - Verilog
  - Many others (vendor specific)
- VHDL and Verilog are IEEE standards
  - They offer portability across different CAD tools and different types of programmable chips.

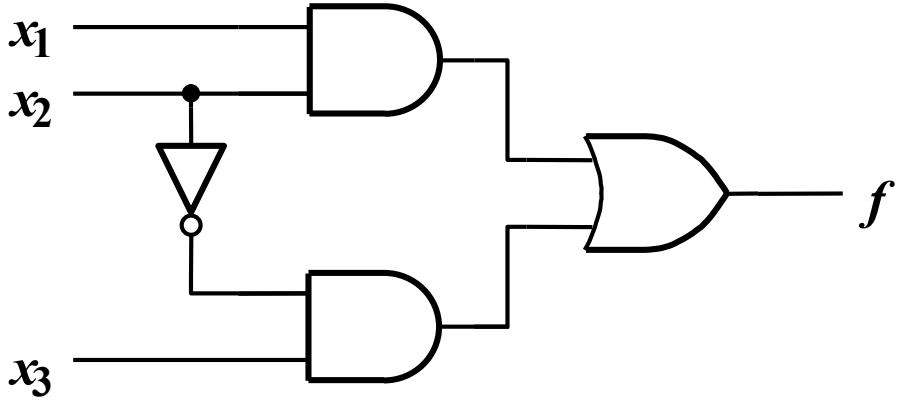
Introduction to Verilog

- The designer writes a logic circuit description in Verilog.
- The Verilog compiler translates this code into simulation form.
- The Verilog code is simulated and tested.
- Synthesis realizes the design on the target hardware (for example, an FPGA).

- Verilog allows a designer to represent circuits in two fundamentally different ways:
  - **Structural** – in terms of gates – how to implement
  - **Behavioral** – in terms of expressions – what to do

Structural Specification (by gates)

A logic circuit is specified in the form of a ***module*** that contains the statements that define the circuit.




```
module example1(x1, x2, x3, f);
  input x1, x2, x3;
  output f;

  and(g, x1, x2);
  not(k, x2);
  and(h, k, x3);
  or(f, g, h);

endmodule
```

Behavioral Specification (what it does)

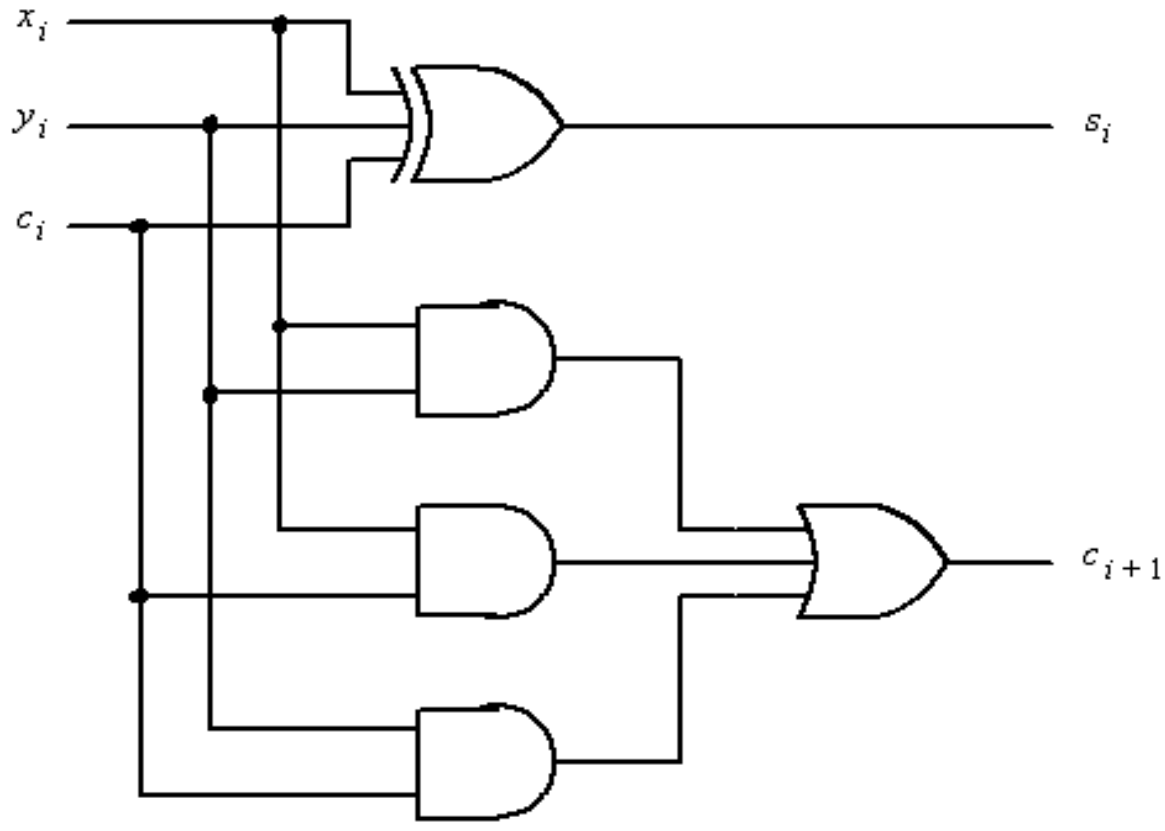
$$f = x_1 x_2 + \overline{x_2} x_3$$

```
module example2(x1, x2, x3, f);
  input x1, x2, x3;
  output f ;

  assign f = (x1 & x2) | (~x2 & x3);

endmodule
```

$$w = \left(x + y'\right) z$$

```
module example3(w, x, y, z);
  input x, y, z;
  output w;

  assign w = (x | ~y) & z;

endmodule
```

Full Adder Circuit



Full Adder – Behavioral Model

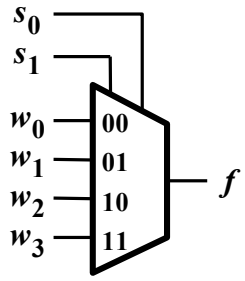
```
module fulladd (Cin, x, y, s, Cout);
    input Cin, x, y;
    output reg s, Cout;

    always @(x, y, Cin)
      {Cout, s} = x + y + Cin;

endmodule
```

The code describes behavior and the Verilog complier implements the details however it considers to be optimum.

Combinational Blocks

- Start with **Multiplexers**
- A **multiplexer (MUX)** circuit has
  - A number of data inputs
  - One or more select inputs
  - One output
- It passes the signal value on one of its data inputs to its output based on the value(s) of the select signal(s).
- Usually, the number of data inputs, *n*, is a power of two. E.g. 2-to-1 multiplexer, 4-to-1 multiplexer, 8-to-1 multiplexer, 16-to-1 multiplexer …

A 4-to-1 Multiplexer

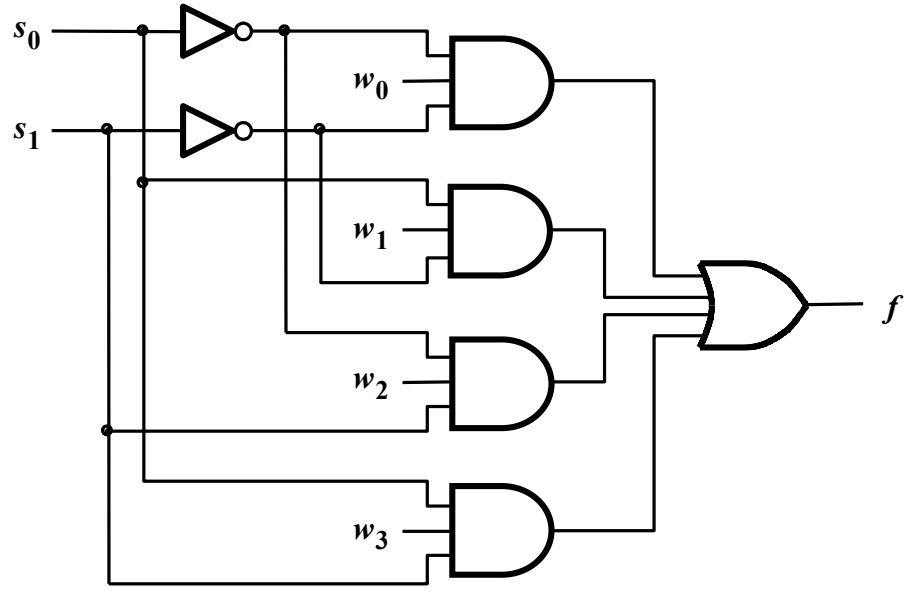


(a) Graphic symbol

s_1	s_0	f
0	0	w_0
0	1	w_1
1	0	w_2
1	1	w_3

(b) Truth table

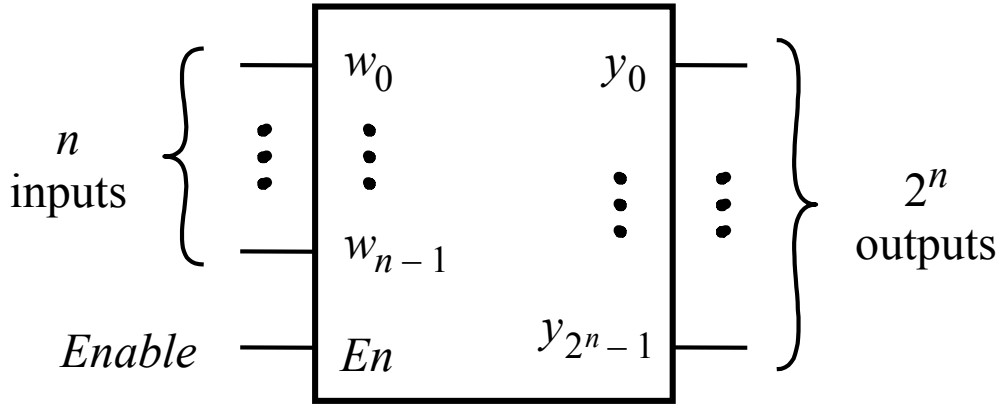
$$f = \bar{s}_1 \bar{s}_0 w_0 + \bar{s}_1 s_0 w_1 + s_1 \bar{s}_0 w_2 + s_1 s_0 w_3$$



(c) Circuit

Decoders

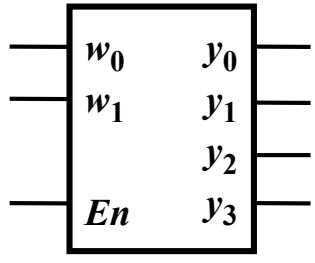
- A binary **decoder** has ***n*** data inputs and 2^n outputs.
- Only one output is asserted at any time (***one-hot encoded***) and each output corresponds to one valuation of the inputs.
- An *enable input* (En) is used to disable the outputs
  - If En=0, none of the decode outputs is asserted
  - If En=1, one of the outputs is asserted according to the valuation of the inputs.



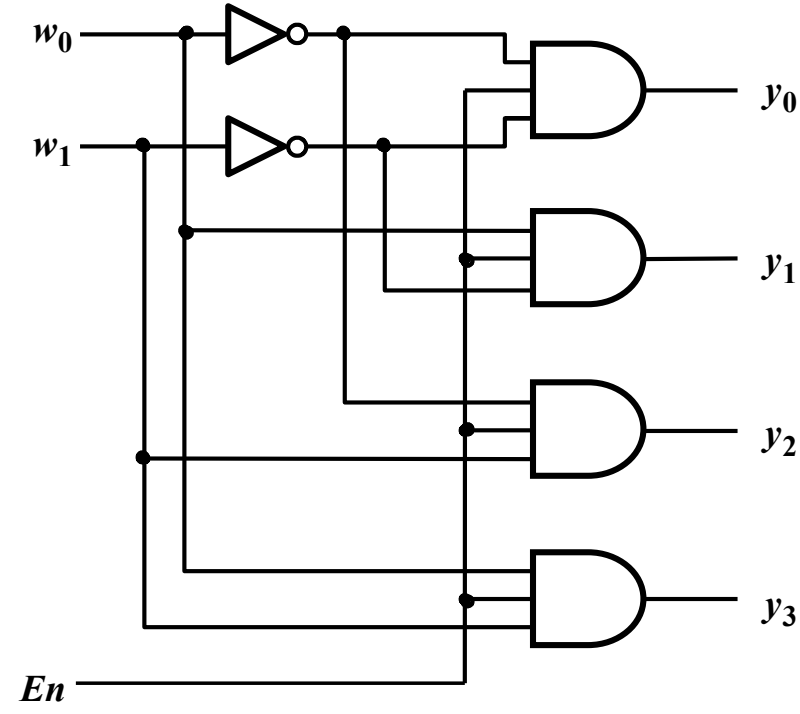
A 2-to-4 Decoder

En	w_1	w_0	y_0	y_1	y_2	y_3
1	0	0	1	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	1	0
1	1	1	0	0	0	1
0	x	x	0	0	0	0

(a) Truth table



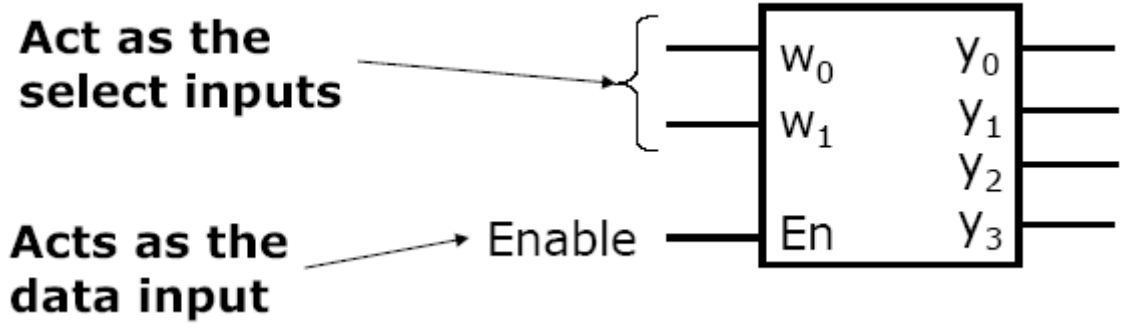
(b) Graphical symbol



(c) Logic circuit

Demultiplexers

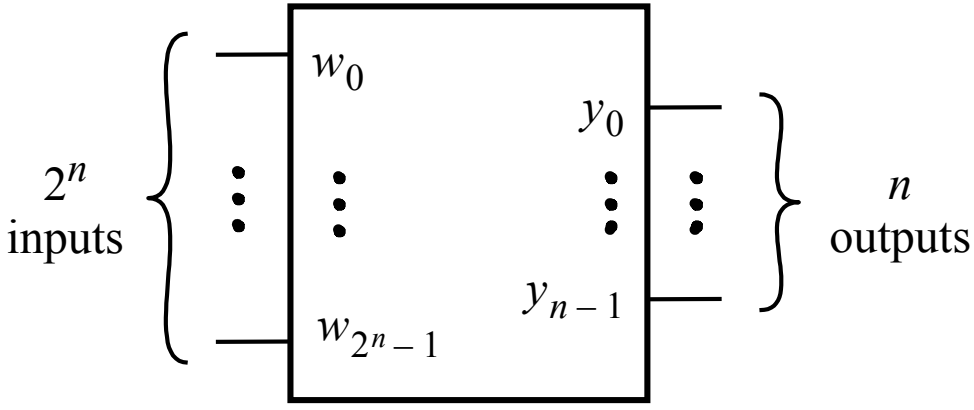
- A multiplexer multiplexed *n* data inputs to a single output.
- A circuit that performs the opposite, placing the value of single data input onto multiple data outputs, is called a ***demultiplexer***.
- A *n*-to-2^n decoder implements a *1*-to-*n* demultiplexer



Encoders

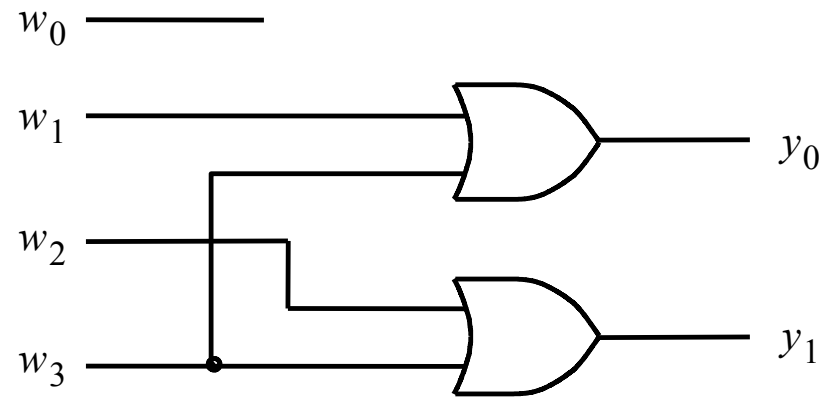
- An **encoder** performs the opposite function of a decoder.
- A **binary encoder** encodes information (data) from 2^n inputs into an n-bit code (output) .
  - Exactly one of the inputs should have a value of 1
  - The outputs represent the binary number that identifies which input is equal to 1.
- Encoders reduce the number of bits needed to represent given information.
- Practical use: transmitting information in digital system.

Encoders - continued



A 2^n-to-n binary encoder.

w_3	w_2	w_1	w_0	y_1	y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1



A 4-to-2 binary encoder.

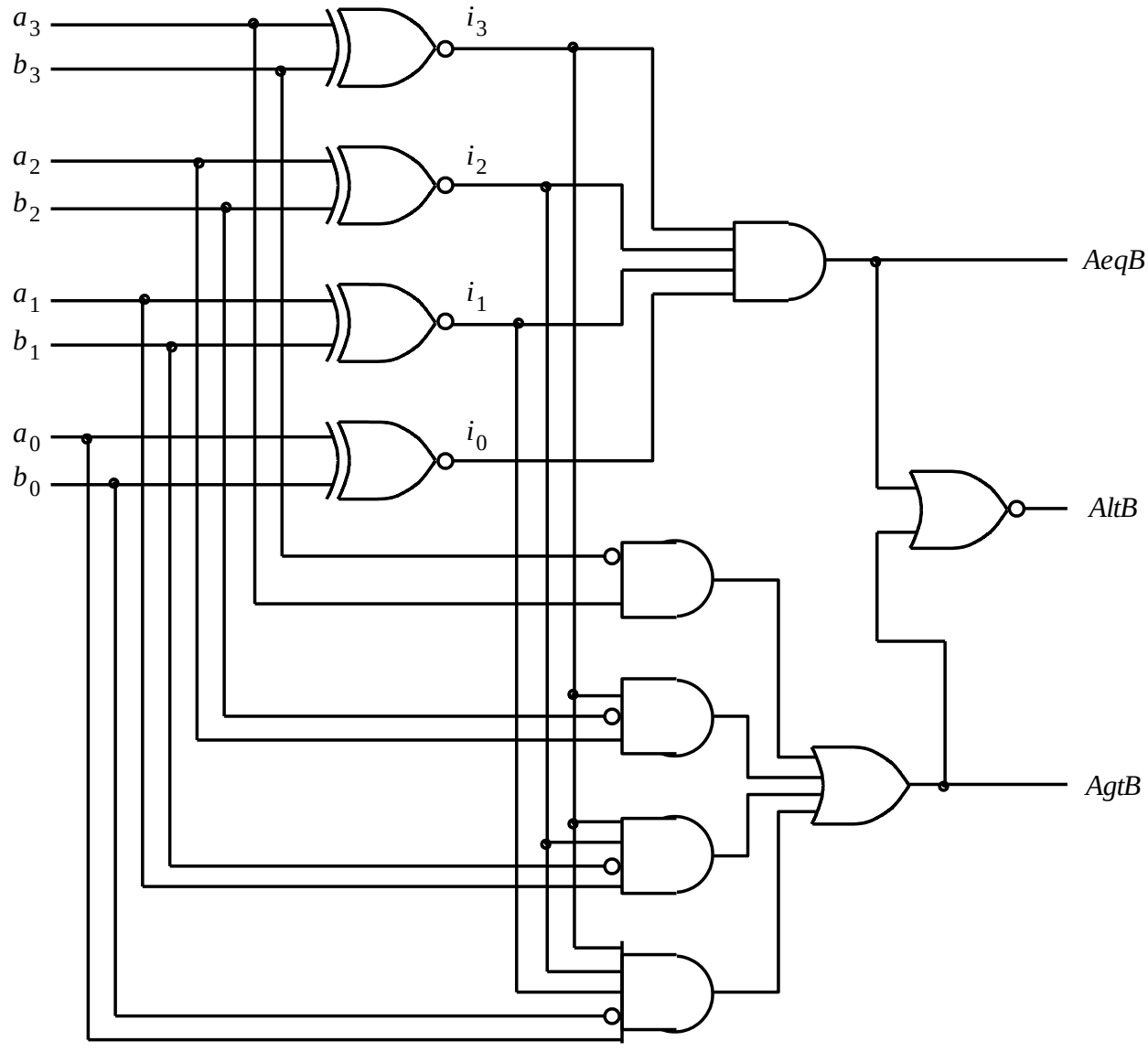
Code Converters

- The purpose of **code converter** circuits is to convert from one type of input encoding to another type of output encoding.
- For example,
  - A 3-to-8 decoder converts from a binary number to a one-hot encoding at the output.
  - A 8-to-3 encoder performs the opposite.
- Many different types of code converter circuits can be constructed.
  - One common example is a BCD-to-7-segment decoder.

Arithmetic Comparison Circuits

- A useful type of arithmetic circuit is called ***comparator*** which compares two ***n***-bit binary numbers.
- For two ***n***-bit numbers A and B, the comparator produces three outputs, called *AeqB*, *AgtB*, and *AltB* .
  - The *AeqB* output is set to 1 if A=B
  - The *AgtB* output is set to 1 if A>B
  - The *AltB* output is set to 1 if A<B
- The desired comparator can be designed by creating a truth table that specifies the three outputs as functions of A and B. However, even for a moderate value of n, **the truth table is large**.

A Four-Bit Comparator Circuit



Combinational vs Sequential Circuits

- Up until now we considered ***combinational circuits*** where the value of each output depends solely on the values of signals applied to the input.

- Another class of circuits are referred to as ***sequential circuits*** where the outputs depend not only on the current inputs, but also the past behavior of the circuit.

Sequential Circuits

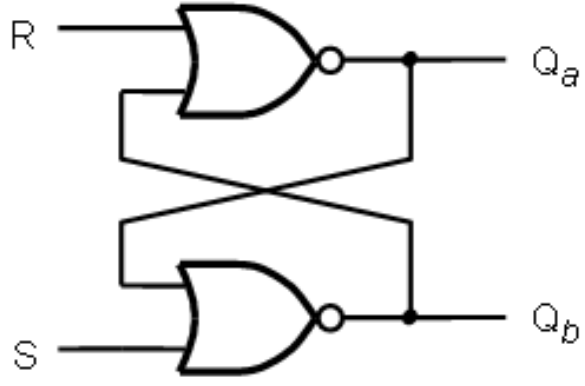
- Sequential circuits include **storage elements** that store the values of logic signals.
- The contents of the storage elements represent the ***state*** of the circuit.
- Input value changes may leave the circuit in the same state or cause it to a new state.
- Over time, the circuit changes through a sequence of states as a result of changes in the inputs.

Summary of Storage Elements

- **Basic Latch:** Feedback connection of NOR gates (active-high Set, Reset inputs) or NAND gates (active-low nSet, nReset inputs). Stores one bit of information.
- **Gated Latch:** A Basic Latch with an additional level sensitive control signal (Clock) that when active allows normal latch operation.
  - Gated SR Latch
  - Gated D Latch
- **Flip-Flop:** A storage element based on a gated latch, with an additional edge sensitive control signal (Clock).
  - D (Data) Flip-Flop
  - T (Toggle) Flip-Flop
  - JK (Set-Reset-Toggle) Flip-Flop

Basic SR Latch

- A more usually way of drawing the previous circuit is shown below.
- The table describes its behavior. Since it does not represent a combinational circuit in which the values of the outputs are determined solely by the current values of the inputs, it is often called a ***characteristic table*** rather than a truth table.



(a) Circuit

S	R	Q_a	Q_b	
0	0	0/1	1/0	(no change)
0	1	0	1	
1	0	1	0	
1	1	0	0	

(b) Characteristic Table

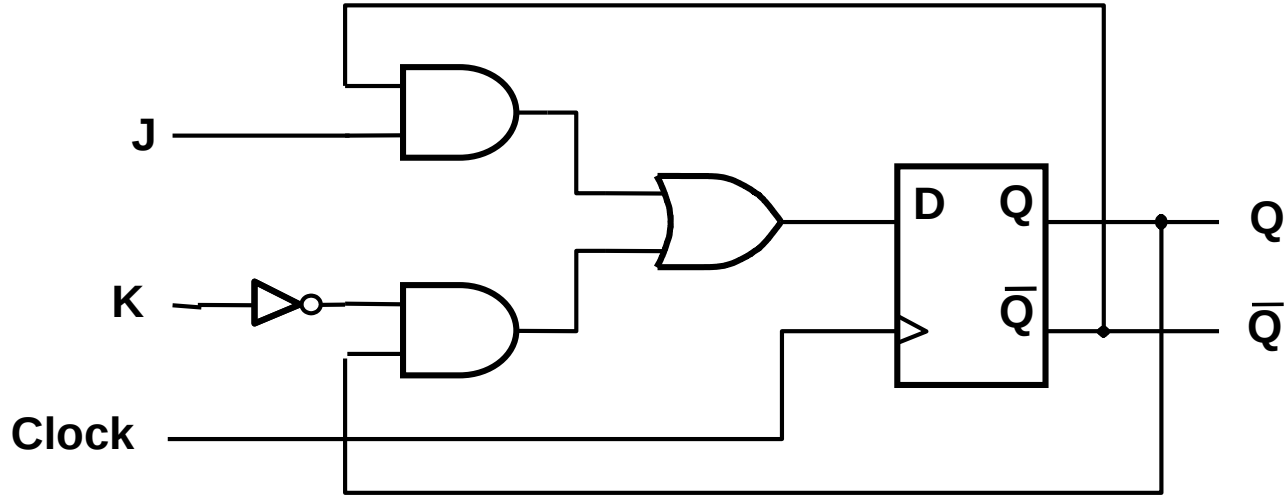
Level vs Edge Sensitive

- Since the output of the D latch is controlled by the level (0 or 1) of the clock input, the latch is said to be ***level sensitive***.
  - All of the latches we have seen have been level sensitive.
- It is possible to design a storage element for which the output only changes at the point in time when the clock changes from one value to another.
- Such circuits are said to be ***edge triggered***.

Flip-Flops

- In the level-sensitive latches, the state of the latch keeps changing according to the values of input signals during the 'active' period of the clock signal.
- There is also a need for storage elements that can change their states **no more than once during one clock cycle.**
- The term ***flip-flop*** denotes a storage element that changes its output at the ***edge*** of a controlling clock signal.

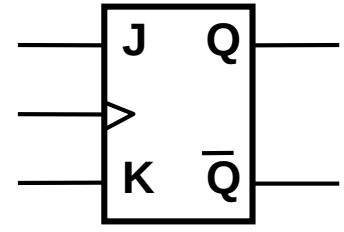
JK Flip-Flop Circuit



(a) Circuit

J	K	Q (t+1)
0	0	Q (t)
0	1	0
1	0	1
1	1	$\overline{Q}$ (t)

(b) Truth table



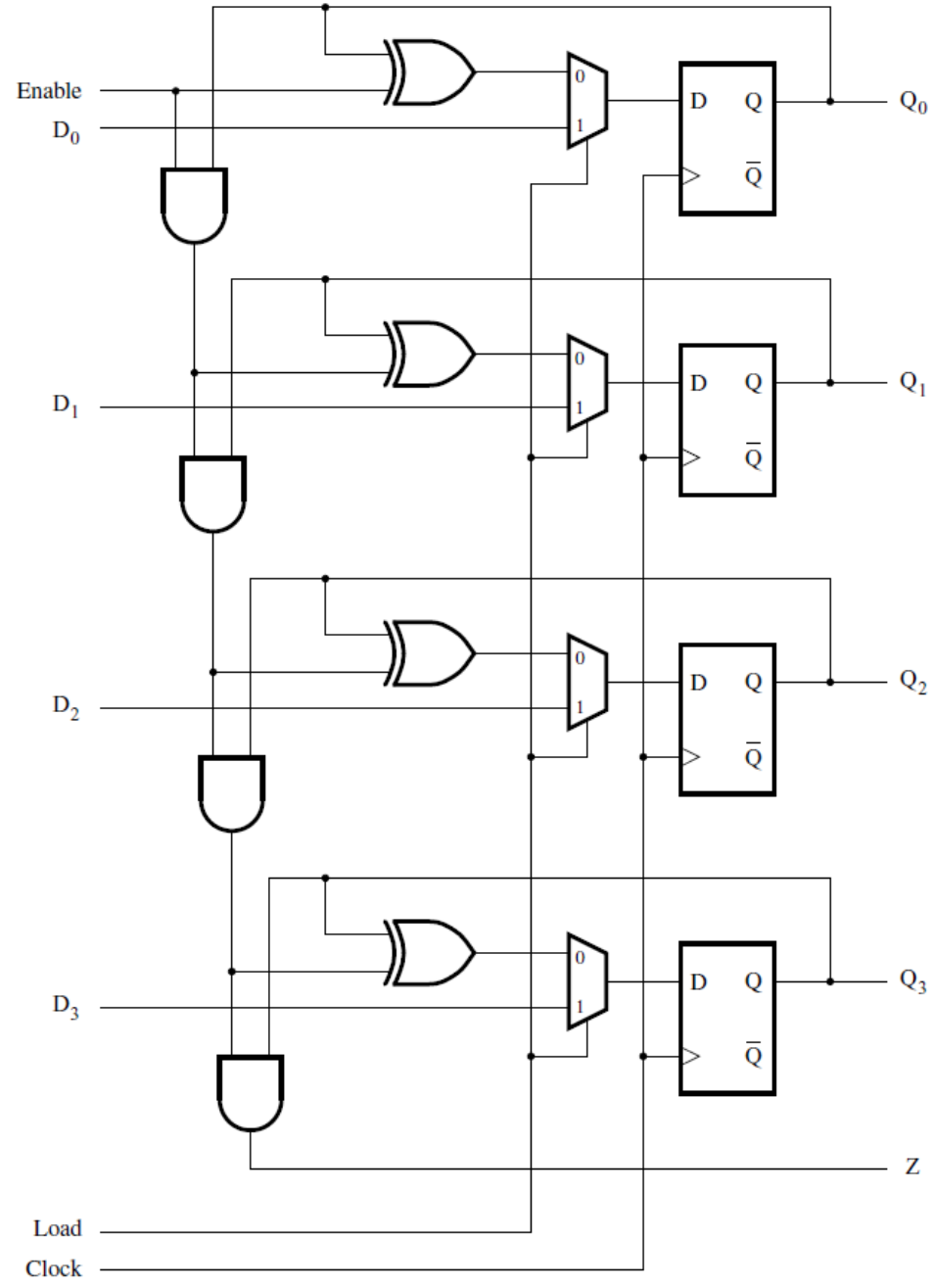
(c) Graphical symbol

Counters

- ***Counters*** are special types of arithmetic circuits that are used for the purpose of counting
  - Circuits that can increment or decrement a count by 1
- Counter circuits are used for many purposes
  - Count the number of occurrences of certain events
  - Generate timing intervals for control of various tasks in a system
  - Track time elapsed between specific events
- It is convenient to introduce counters built with ***T flip-flops*** because the toggle feature is naturally suited for implementing the counting operation.

Four Bit Up-Counter with Parallel Load

- MUXs select either the counting control signals or load value D.



Four-Bit Up-Counter with Parallel Load
(and asynchronous reset)

```
module upcount (R, Resetn, Clock, E, L, Q);
    input [3:0] R;
    input Resetn, Clock, E, L;
    output reg [3:0] Q;

    always @(negedge Resetn, posedge Clock)
        if (!Resetn)
            Q <= 0;
        else if (L)
            Q <= R;
        else if (E)
            Q <= Q + 1;

endmodule
```