Computer organization

Levels of abstraction

	Assembler Simulator	Applications
	C C++ Java	High-level language
		SOFTWARE
	add lw ori	Assembly language
Goal	0000 0001 0000 1001 0101	Machine instructions/Data
		HARDWARE
	CPU Memory I/O	Functional units
CMSC311	Multiplexor Decoder Adder	Components
	Combinational Sequential	Circuits
You are here	AND OR XOR	Gates
	Transistors Wires	Electronics
	Electrons	Atomic units

Gates

Gates: NOT Mr. Bill!

Basic building blocks for circuits

Implement boolean functions in hardware

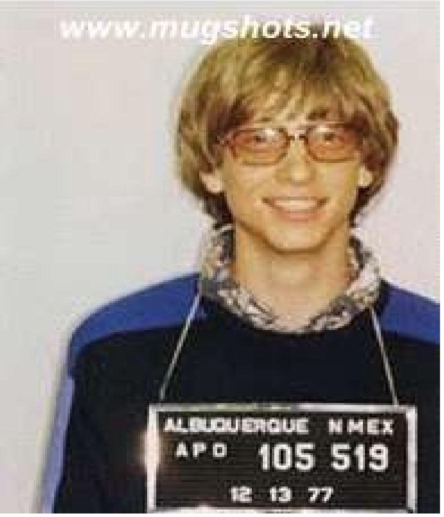
- Computer engineering: how to build it physically
- Computer organization: how to design it logically

Combinational circuits

- Output depends only on input

Sequential circuits

- Output depends on input AND current state



Gates: truth tables

		NOT		AND	OR	XOR	NOR	NAND	XNOR
a	b	~a	~b	a & b	a | b	a ^ b	~(a | b)	~(a & b)	~(a ^ b)
0	0	1	1	0	0	0	1	1	1
0	1	1	0	0	1	1	0	1	0
1	0	0	1	0	1	1	0	1	0
1	1	0	0	1	1	0	0	0	1

How many possible boolean functions of 2 variables?

Depends on number of outputs

4 possible inputs --> 16 possible outputs of 1 bit each

Gates: truth tables

	Unsigned Binary						Function Name	Gates
Inputs		a	0	0	1	1		
		b	0	1	0	1		
Outputs	0		0	0	0	0	FALSE	
	1		0	0	0	1	AND	✓
	2		0	0	1	0	a & ~b	
	3		0	0	1	1	a	
	4		0	1	0	0	~a & b	
	5		0	1	0	1	b	
	6		0	1	1	0	XOR	✓
	7		0	1	1	1	OR	✓
	8		1	0	0	0	NOR	✓
	9		1	0	0	1	XNOR	✓
	10		1	0	1	0	~b	
	11		1	0	1	1	a | ~b	
	12		1	1	0	0	~a	
	13		1	1	0	1	~a | b	
	14		1	1	1	0	NAND	✓
	15		1	1	1	1	TRUE	

Gates: Inverter

Inverter: implements NOT function

Also known as "negation" or "complement"

Input: 1 bit

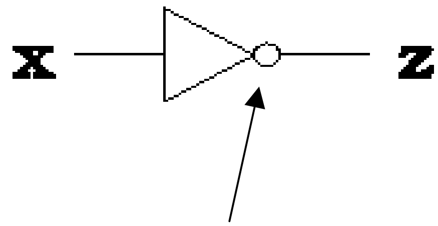
Output: 1 bit

Truth table:

Input	Output
x	z
0	1
1	0

`z = ~x`

Symbol:



Circle indicates negation

Other notation:

`z = x'`

$z = \overline{x}$

`z = \x`

Gates: AND

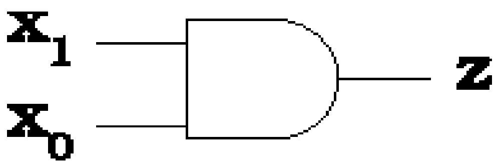
AND gate: implements AND function
Input: 2 bits
Output: 1 bit

Truth table:

Input		Output
x_0	x_1	z
0	0	0
0	1	0
1	0	0
1	1	1

$$z = x_0 \,\&\, x_1$$

Symbol:



Other notation:

$$z = \mathrm{AND}\ (x_0,\ x_1)$$
$$z = x_0 * x_1$$
$$z = x_0\ x_1$$

Properties:

symmetric: $x * y = y * x$

associative: $(x * y) * z = x * (y * z)$

n inputs:

$$\mathrm{AND}_n\ (x_0,\ x_1,\ .\ .\ .\ ,\ x_n) = x_0 * x_1 * .\ .\ .\ x_n$$

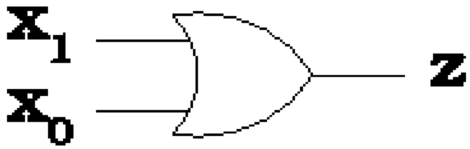
Gates: OR

OR gate: implements OR function
Input: 2 bits
Output: 1 bit

Truth table:

Input		Output
x_0	x_1	z
0	0	0
0	1	1
1	0	1
1	1	1

Symbol:



$$z = x_0 \mid x_1$$

Other notation:

$$z = \mathrm{OR}\ (x_0,\ x_1)$$
$$z = x_0 + x_1$$

Properties:

symmetric: $x + y = y + x$

associative: $(x + y) + z = x + (y + z)$

n inputs:

$$\mathrm{OR}_n\ (x_0,\ x_1,\ .\ .\ .\ ,\ x_n) = x_0 + x_1 + .\ .\ .\ x_n$$

Gates: NAND

NAND gate: implements NAND (negated AND) function

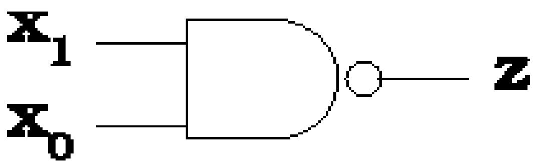
Input: 2 bits

Output: 1 bit

Truth table:

Input		Output
x_0	x_1	z
0	0	1
0	1	1
1	0	1
1	1	0

Symbol:



$$z = x_0 \text{ NAND } x_1$$

Properties:

symmetric: $x \text{ NAND } y = y \text{ NAND } x$

not associative

n inputs:

$$\text{NAND}_n\ (x_0,\ x_1,\ .\ .\ .\ ,\ x_n) = \text{NOT}\ (x_0 * x_1 * .\ .\ .\ x_n)$$

Gates: NOR

NOR gate: implements NOR (negated OR) function

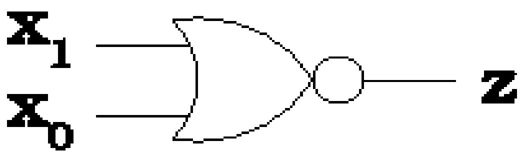
Input: 2 bits

Output: 1 bit

Truth table:

Input		Output
x_0	x_1	z
0	0	1
0	1	0
1	0	0
1	1	0

Symbol:



$$z = x_0 \text{ NOR } x_1$$

Properties:

symmetric: $x \text{ NOR } y = y \text{ NOR } x$

not associative

n inputs:

$$\text{NOR}_n (x_0, x_1, \ldots, x_n) = \text{NOT } (x_0 + x_1 + \ldots x_n)$$

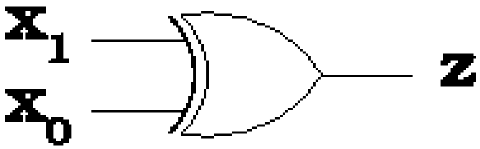
Gates: XOR

XOR gate: implements exclusive-OR function
Input: 2 bits
Output: 1 bit

Truth table:

Input		Output
x_0	x_1	z
0	0	0
0	1	1
1	0	1
1	1	0

Symbol:



$$z = x_0 \wedge x_1$$

Properties:

symmetric: $x \wedge y = y \wedge x$

associative: $(x \wedge y) \wedge z = x \wedge (y \wedge z)$

n inputs:

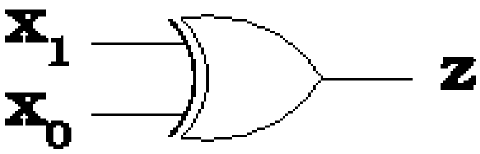
$$XOR_n\ (x_0, x_1, \ldots, x_n) = x_0 \wedge x_1 \wedge \ldots x_n$$

Gates: XOR properties

Truth table:

Input		Output
x_0	x_1	z
0	0	0
0	1	1
1	0	1
1	1	0

Symbol:



XOR can be expressed in terms of AND, OR, NOT:

`x XOR y == ( x AND (NOT y) ) OR ( (NOT x) AND y )`

(If x is true, y must be false, and vice versa.)

x_0 ^ x_1 ^ . . . x_n **is true if the number of true values is odd, and false if the number of true values is even. Why?**

x_0 ^ x_1 ^ . . . x_n == (x_0 + x_1 + . . . x_n) % 2

XOR is the same as the sum modulo 2

`x ^ 0 = x` **XORing with 0 gives you back the same number (identity)**

`x ^ 1 = ~x` **XORing with 1 gives you the complement**

`x ^ x = 0` **XORing a number with itself gives 0**

Gates: XOR properties

More XOR tricks (amaze your friends!):

Classic swap problem (early CMSC 106)

```
temp = x;
x = y;
y = temp;
```

Using XOR:

```
x = x ^ y ;
y = x ^ y ;
x = x ^ y ;
```

Let `x0` be the original value of `x`, `y0` be the original value of `y`:

`x = x ^ y = x0 ^ y0`	
`y = x ^ y = (x0 ^ y0) ^ y0`	**Substitute for x**
`  = x0 ^ (y0 ^ y0)`	**Associative property**
`  = x0 ^ 0`	**x ^ x = 0**
`  = x0`	**Identity**
`x = x ^ y = (x0 ^ y0) ^ x0`	**Substitute for x and y**
`  = (x0 ^ x0) ^ y0`	**Associative, symmetric properties**
`  = 0 ^ y0`	**x ^ x = 0**
`  = y0`	**Identity**

What other operator is a less-safe way of doing this?

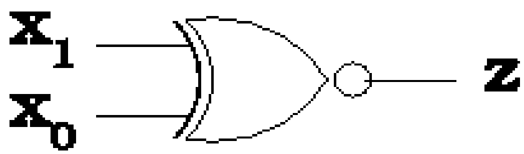
Gates: XNOR

XNOR gate: implements XNOR (negated exclusive-OR) function
Input: 2 bits
Output: 1 bit

Truth table:

Input		Output
x_0	x_1	z
0	0	1
0	1	0
1	0	0
1	1	1

Symbol:



$$z = x_0 \text{ XNOR } x_1 = \sim(x_0 \wedge x_1)$$

Properties:

symmetric: x XNOR y = y XNOR x

associative: (x XNOR y) XNOR z = x XNOR (y XNOR z)

n inputs:

$$\text{XNOR}_n\ (x_0, x_1, \ldots, x_n) = x_0 \text{ XNOR } x_1 \text{ XNOR } \ldots x_n$$

Gates: Buffer

Buffer: implements equality function

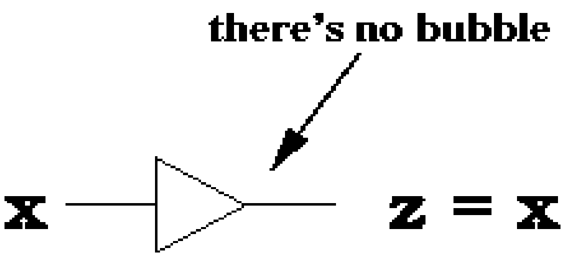
Input: 1 bit

Output: 1 bit

Truth table:

Input x	Output z
0	0
1	1

Symbol:



This doesn't look very interesting at all!

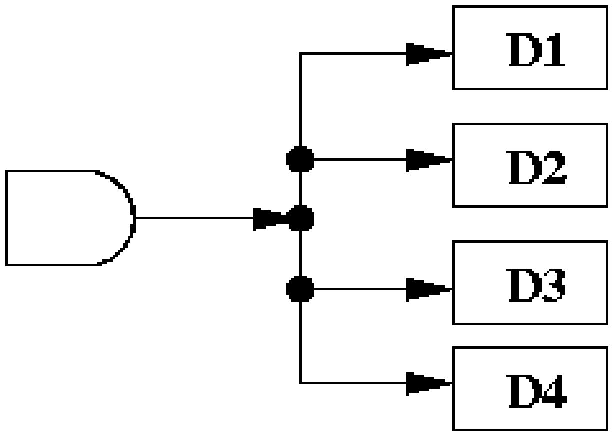
There is a practical reason for it, however:

Circuits use electrical signals: 0 and 1 are represented by voltage.

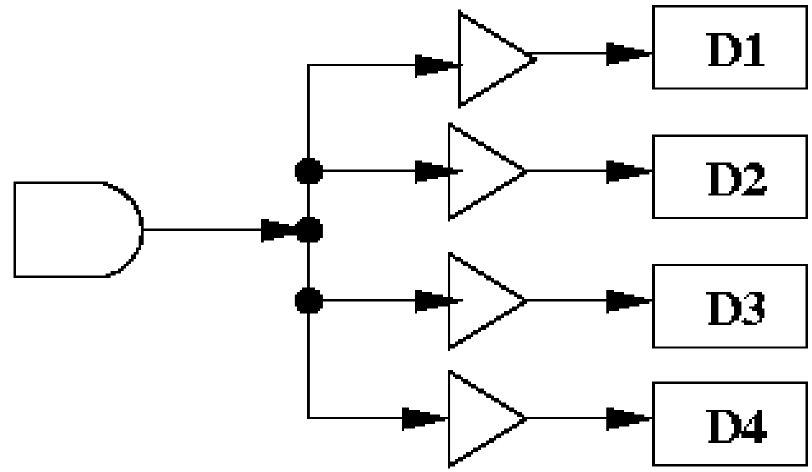
If current is too low, it's hard to measure voltage accurately.

"Fan out" (number of devices) reduces amount of current.

If the current from the AND gate is distributed equally, then each device gets 1/4 the current.



A buffer can be used to "boost" the current back to the right level:



The buffer (like all other gates) is an active device; it requires power input to maintain current and voltage.

That's all EE stuff, and we're programmers. Why should we care about that?

Gates: Tri-State Buffer

A tri-state buffer acts like a valve: controls flow of current.

Input: 2 bits

Output: 1 bit

Truth table:

Input		Output
c	x	z
0	0	Z
0	1	Z
1	0	0
1	1	1

no current

simplified:

active-high

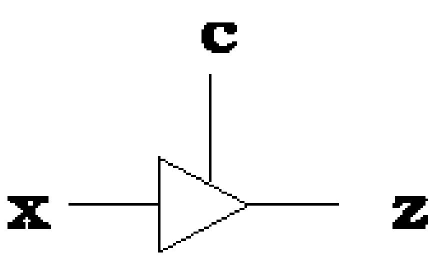
Input	Output
c	z
0	Z
1	x

active-low

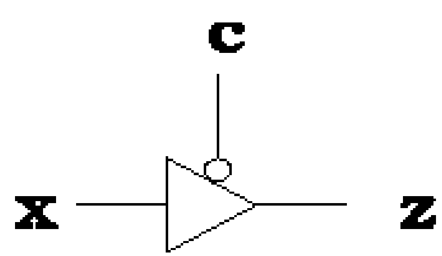
Input	Output
c	z
0	x
1	Z

When c = 1, the output is equal to x, otherwise there is no output.

Active-low: Output is x when c = 0.



tri–state buffer with active high control



tri–state buffer with active low control

Circuits

Gates may be connected to build circuits

Valid combinational circuits

- The output of a gate may only be attached to the input of another gate.
  - Think of this as a directed edge from output to input.
- There must be no cycles in the circuit (directed graph).
- Although the output of a gate may be attached to more than one input,
  - an input may not have two different outputs attached to it
  - (This would create conflicting input signals.)
- Each input of a gate must come from either the output of another gate or a source.
  - Source: something which generates either a constant 0 or 1.

Gate delay

- Output takes some small amount of time before it changes.
- Information can travel at most, at the speed of light.
- Gate delay limits how fast the inputs can change and
  - the output can still have meaningful values.