Boolean logic

Introduction to Computer

Yung-Yu Chuang

*with slides by Sedgewick & Wayne (*introcs.cs.princeton.edu*), Nisan & Schocken (*www.nand2tetris.org*) and Harris & Harris (DDCA)*

Boolean Algebra

Based on symbolic logic, designed by George Boole
Boolean variables take values as 0 or 1.
Boolean expressions created from:
  NOT, AND, OR

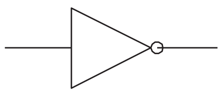
George Boole
1815 - 1864

NOT

$\neg X$ $\overline{X}$ X'

x	Not
0	1
1	0

Digital gate diagram for NOT:



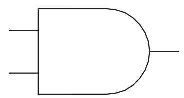
NOT

AND

$X \wedge Y$ $X \cdot Y$ XY

x	y	And
0	0	0
0	1	0
1	0	0
1	1	1

Digital gate diagram for AND:



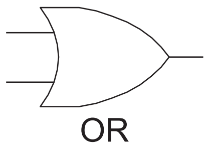
AND

OR

X∨Y   X+Y

x	y	Or
0	0	0
0	1	1
1	0	1
1	1	1

Digital gate diagram for OR:



Operator Precedence

Examples showing the order of operations:
NOT > AND > OR

Expression	Order of Operations
$\neg X \vee Y$	NOT, then OR
$\neg(X \vee Y)$	OR, then NOT
$X \vee (Y \wedge Z)$	AND, then OR

Use parentheses to avoid ambiguity

Defining a function

Description: square of x minus 1
Algebraic form : x^2-1
Enumeration:

x	f(x)
1	0
2	3
3	8
4	15
5	24
:	:

Defining a function

Description: number of days of the x-th month of a non-leap year
Algebraic form: ?
Enumeration:

x	f(x)
1	31
2	28
3	31
4	30
5	31
6	30
7	31
8	31
9	30
10	31
11	30
12	31

Truth Table

Truth table.

Systematic method to describe Boolean function.
One row for each possible input combination.
N inputs $\Rightarrow$ 2^N rows.

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

AND truth table

Proving the equivalence of two functions

Prove that $x^2-1=(x+1)(x-1)$

Using algebra: (you need to follow some rules)
$(x+1)(x-1) = x^2+x-x-1= x^2-1$

Using enumeration:

x	(x+1)(x-1)	x^2-1
1	0	0
2	3	3
3	8	8
4	15	15
5	24	24
:	:	:

Important laws

$x + 1 = 1$
$x + 0 = x$
$x + \overline{x} = 1$

$x \cdot 1 = x$
$x \cdot 0 = 0$
$x \cdot \overline{x} = 0$

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

$x \cdot y = y \cdot x$
$x \cdot (y \cdot z) = (x \cdot y) \cdot z$

$x \cdot (y+z) = xy + xz$

DeMorgan Law

$\overline{x \cdot y} = \overline{x} + \overline{y}$

Simplifying Boolean expressions

Example 1

- $Y = AB + \overline{A}B$

Simplifying Boolean expressions

Example 1

- $Y = AB + \overline{A}B$

  $= B(A + \overline{A})$

  $= B(1)$

  $= B$

Simplifying Boolean expressions

Example 2

- $Y = A(AB + ABC)$

Simplifying Boolean expressions

Example 2

- $Y = A(AB + ABC)$

  $= A(AB(1 + C))$

  $= A(AB(1))$

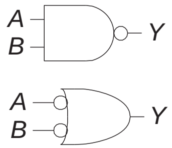
  $= A(AB)$

  $= (AA)B$

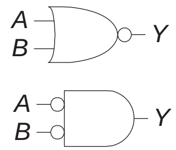
  $= AB$

DeMorgan's Theorem

- $Y = \overline{AB} = \overline{A} + \overline{B}$



- $Y = \overline{A + B} = \overline{A} \cdot \overline{B}$



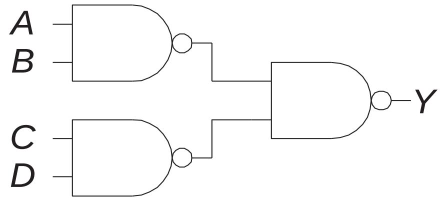
Bubble pushing

- **Backward:**
  - Body changes
  - Adds bubbles to inputs

- **Forward:**
  - Body changes
  - Adds bubble to output

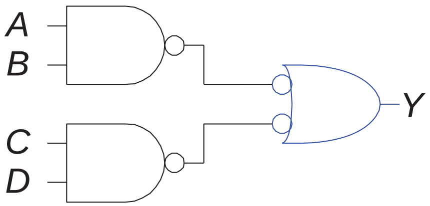
Bubble pushing

- What is the Boolean expression for this circuit?



Bubble pushing

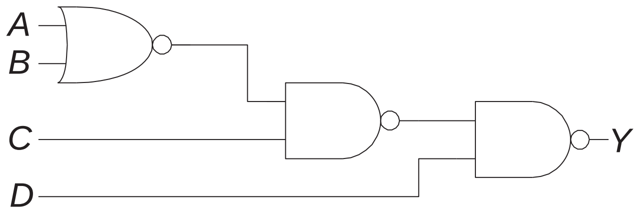
- What is the Boolean expression for this circuit?



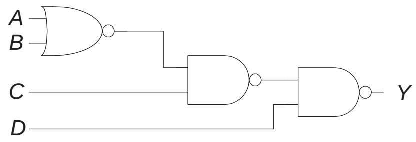
$$Y = AB + CD$$

Bubble pushing rules

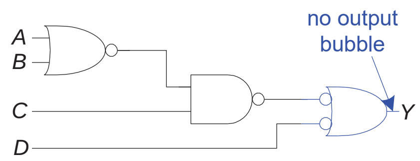
- Begin at output, then work toward inputs
- Push bubbles on final output back
- Draw gates in a form so bubbles cancel



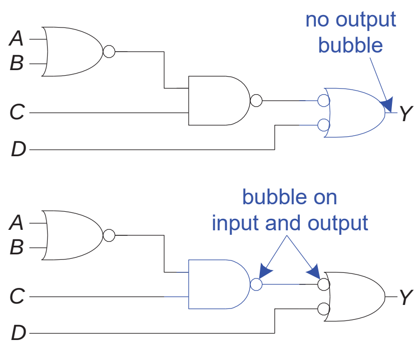
Bubble pushing example



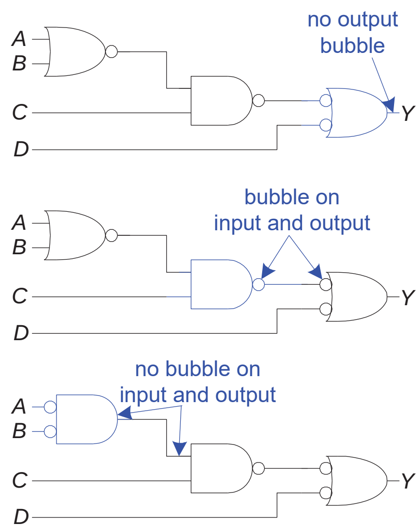
Bubble pushing example



Bubble pushing example



Bubble pushing example



$$Y = \overline{A}\overline{B}C + \overline{D}$$

Truth Tables

A Boolean function has one or more Boolean inputs, and returns a single Boolean output.
A truth table shows all the inputs and outputs of a Boolean function

Example: $\neg x \vee y$

x	y	$\neg x \vee y$
0	0	
0	1	
1	0	
1	1	

Truth Tables

A Boolean function has one or more Boolean inputs, and returns a single Boolean output.
A truth table shows all the inputs and outputs of a Boolean function

Example: $\neg x \vee y$

x	y	$\neg x$	$\neg x \vee y$
0	0		
0	1		
1	0		
1	1		

Truth Tables

A Boolean function has one or more Boolean inputs, and returns a single Boolean output.
A truth table shows all the inputs and outputs of a Boolean function

Example: $\neg x \vee y$

x	y	$\neg x$	$\neg x \vee y$
0	0	1	1
0	1	1	1
1	0	0	0
1	1	0	1

Truth Tables

Example: $X \wedge \neg Y$

x	y	$\neg y$	$x \wedge \neg y$
0	0		
0	1		
1	0		
1	1		

Truth Tables

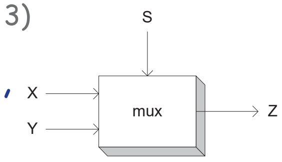
Example: $X \wedge \neg Y$

x	y	$\neg y$	$x \wedge \neg y$
0	0	1	0
0	1	0	0
1	0	1	1
1	1	0	0

Truth Tables (3 of 3)

When S=0, return X; otherwise, return Y.

Example: $(Y \wedge S) \vee (X \wedge \neg S)$



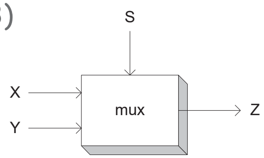
Two-input multiplexer

S	X	Y	Y∧S	¬S	X∧¬S	(Y ∧ S) ∨ (X ∧ ¬S)
0	0	0				
0	0	1				
0	1	0				
0	1	1				
1	0	0				
1	0	1				
1	1	0				
1	1	1				

Truth Tables (3 of 3)

When S=0, return X; otherwise, return Y.

Example: $(Y \wedge S) \vee (X \wedge \neg S)$



Two-input multiplexer

S	X	Y	Y∧S	¬S	X∧¬S	(Y ∧ S) ∨ (X ∧ ¬S)
0	0	0	0	1	0	0
0	0	1	0	1	0	0
0	1	0	0	1	1	1
0	1	1	0	1	1	1
1	0	0	0	0	0	0
1	0	1	1	0	0	1
1	1	0	0	0	0	0
1	1	1	1	0	0	1

Truth Table for Functions of 2 Variables

- 2 variables lead to four possible combinations
- A 2-variable function f has to define four values

x	y	f
0	0	V_{00}
0	1	V_{01}
1	0	V_{10}
1	1	V_{11}

Truth Table for Functions of 2 Variables

Truth table.

16 Boolean functions of 2 variables. every 4-bit value represents one

x	y	ZERO	AND		x		y	XOR	OR
0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

Truth table for all Boolean functions of 2 variables

x	y	NOR	EQ	y'		x'		NAND	ONE
0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

Truth table for all Boolean functions of 2 variables

All Boolean functions of 2 variables

Function	x	0	0	1	1
	y	0	1	0	1
Constant 0	0	0	0	0	0
And	$x \cdot y$	0	0	0	1
x And Not y	$x \cdot \bar{y}$	0	0	1	0
x	x	0	0	1	1
Not x And y	$\bar{x} \cdot y$	0	1	0	0
y	y	0	1	0	1
Xor	$x \cdot \bar{y} + \bar{x} \cdot y$	0	1	1	0
Or	$x + y$	0	1	1	1
Nor	$\overline{x+y}$	1	0	0	0
Equivalence	$x \cdot y + \bar{x} \cdot \bar{y}$	1	0	0	1
Not y	$\bar{y}$	1	0	1	0
If y then x	$x + \bar{y}$	1	0	1	1
Not x	$\bar{x}$	1	1	0	0
If x then y	$\bar{x} + y$	1	1	0	1
Nand	$\overline{x \cdot y}$	1	1	1	0
Constant 1	1	1	1	1	1

Truth Table for Functions of 3 Variables

Truth table.

16 Boolean functions of 2 variables. every 4-bit value represents one
256 Boolean functions of 3 variables. every 8-bit value represents one
2^(2^n) Boolean functions of n variables! every 2^n-bit value represents one

x	y	z	AND	OR	MAJ	ODD
0	0	0	0	0	0	0
0	0	1	0	1	0	1
0	1	0	0	1	0	1
0	1	1	0	1	1	0
1	0	0	0	1	0	1
1	0	1	0	1	1	0
1	1	0	0	1	1	0
1	1	1	1	1	1	1

some functions of 3 variables

Sum-of-Products

Sum-of-products. Systematic procedure for representing a Boolean function using AND, OR, NOT. (proves that { AND, OR, NOT } are universal)

- Form AND term for each 1 in Boolean function.
- OR terms together.

x	y	z	MAJ
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Sum-of-Products

Sum-of-products. Systematic procedure for representing a Boolean function using AND, OR, NOT.

proves that { AND, OR, NOT } are universal

- Form AND term for each 1 in Boolean function.
- OR terms together.

x	y	z	MAJ
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

expressing MAJ using sum-of-products

Sum-of-Products

Sum-of-products. Systematic procedure for representing a Boolean function using AND, OR, NOT.

proves that { AND, OR, NOT } are universal

- Form AND term for each 1 in Boolean function.
- OR terms together.

x	y	z	MAJ
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

expressing MAJ using sum-of-products

Sum-of-Products

Sum-of-products. Systematic procedure for representing a Boolean function using AND, OR, NOT.

proves that { AND, OR, NOT } are universal

- Form AND term for each 1 in Boolean function.
- OR terms together.

x	y	z	MAJ	
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	1	0
1	1	0	1	0
1	1	1	1	0

expressing MAJ using sum-of-products

Sum-of-Products

Sum-of-products. Systematic procedure for representing a Boolean function using AND, OR, NOT.

proves that { AND, OR, NOT } are universal

- Form AND term for each 1 in Boolean function.
- OR terms together.

x	y	z	MAJ	x'yz
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	1	0
1	1	0	1	0
1	1	1	1	0

expressing MAJ using sum-of-products

Sum-of-Products

Sum-of-products. Systematic procedure for representing a Boolean function using AND, OR, NOT.

proves that { AND, OR, NOT } are universal

Form AND term for each 1 in Boolean function.
OR terms together.

x	y	z	MAJ	x'yz	xy'z
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	1	1	0
1	0	0	0	0	0
1	0	1	1	0	1
1	1	0	1	0	0
1	1	1	1	0	0

expressing MAJ using sum-of-products

Sum-of-Products

Sum-of-products. Systematic procedure for representing a Boolean function using AND, OR, NOT.

proves that { AND, OR, NOT } are universal

Form AND term for each 1 in Boolean function.
OR terms together.

x	y	z	MAJ	x'yz	xy'z	xyz'	xyz
0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	1	0	0	0	0	0	0
0	1	1	1	1	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	0	1	0	0
1	1	0	1	0	0	1	0
1	1	1	1	0	0	0	1

expressing MAJ using sum-of-products

Sum-of-Products

Sum-of-products. Systematic procedure for representing a Boolean function using AND, OR, NOT.

proves that { AND, OR, NOT } are universal

Form AND term for each 1 in Boolean function.
OR terms together.

x	y	z	MAJ	x'yz	xy'z	xyz'	xyz	x'yz + xy'z + xyz' + xyz
0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0
0	1	1	1	1	0	0	0	1
1	0	0	0	0	0	0	0	0
1	0	1	1	0	1	0	0	1
1	1	0	1	0	0	1	0	1
1	1	1	1	0	0	0	1	1

expressing MAJ using sum-of-products

Universality of AND, OR, NOT

Fact. Any Boolean function can be expressed using AND, OR, NOT.

{ AND, OR, NOT } are universal.
Ex: XOR(x,y) = xy' + x'y.

Notation	Meaning
x'	NOT x
x y	x AND y
x + y	x OR y

Expressing XOR Using AND, OR, NOT

x	y	x'	y'	x'y	xy'	x'y + xy'	x XOR y
0	0	1	1	0	0	0	0
0	1	1	0	1	0	1	1
1	0	0	1	0	1	1	1
1	1	0	0	0	0	0	0

Universality of AND, OR, NOT

Fact. Any Boolean function can be expressed using AND, OR, NOT.

{ AND, OR, NOT } are universal.

Ex: XOR(x,y) = xy' + x'y.

Notation	Meaning
x'	NOT x
x y	x AND y
x + y	x OR y

Expressing XOR Using AND, OR, NOT

x	y	x'	y'	x'y	xy'	x'y + xy'	x XOR y
0	0	1	1	0	0	0	0
0	1	1	0	1	0	1	1
1	0	0	1	0	1	1	1
1	1	0	0	0	0	0	0

Exercise. Show {AND, NOT}, {OR, NOT}, {NAND}, {NOR} are universal.

Hint. DeMorgan's law: (x'y')' = x + y.

{AND, NOT} is universal

{NAND} is universal

From Math to Real-World implementation

We can implement any Boolean function using NAND gates only.

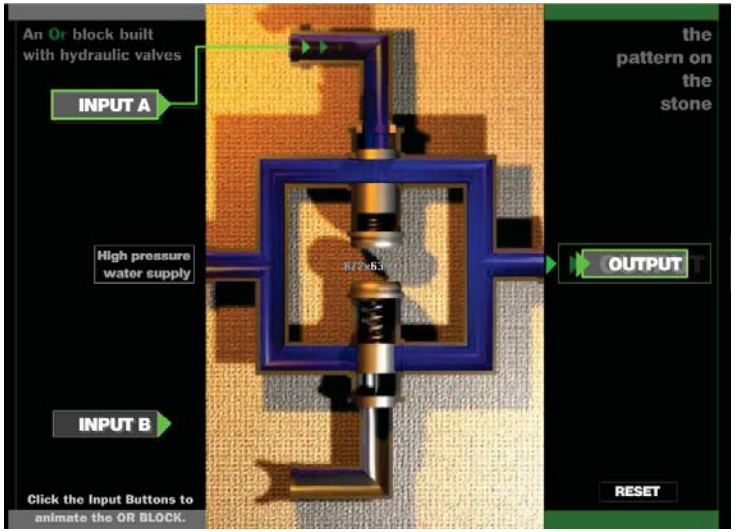
We talk about abstract Boolean algebra (logic) so far.

Is it possible to realize it in real world?

The technology needs to permit switching and conducting. It can be built using magnetic, optical, biological, hydraulic and pneumatic mechanism.

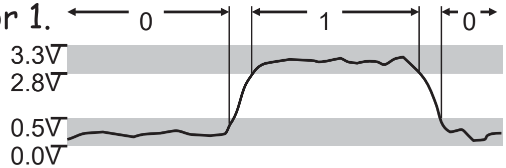
Implementation of gates

Fluid switch
(http://www.cs.princeton.edu/introcs/lectures/fluid-computer.swf)



Digital Circuits

What is a digital system?
Analog: signals vary continuously.
Digital: signals are 0 or 1.

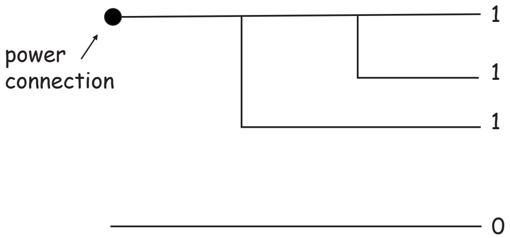


Why digital systems?
Accuracy and reliability.
Staggeringly fast and cheap.

Basic abstractions.
On, off.
Wire: propagates on/off value.
Switch: controls propagation of on/off values through wires.

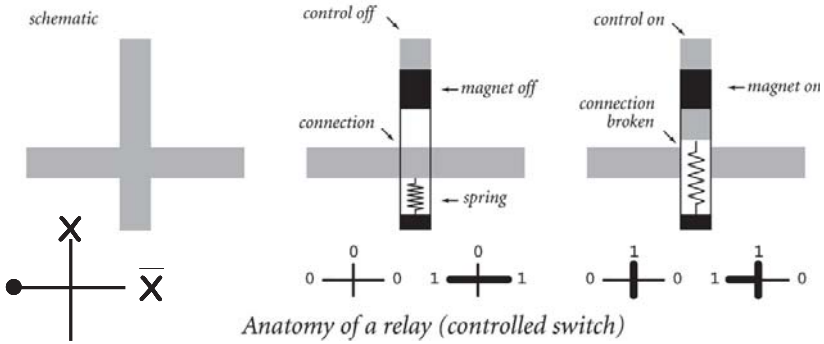
Wires

Wires.
On (1): connected to power.
Off (0): not connected to power.
If a wire is connected to a wire that is on, that wire is also on.
Typical drawing convention: "flow" from top, left to bottom, right.



Controlled Switch

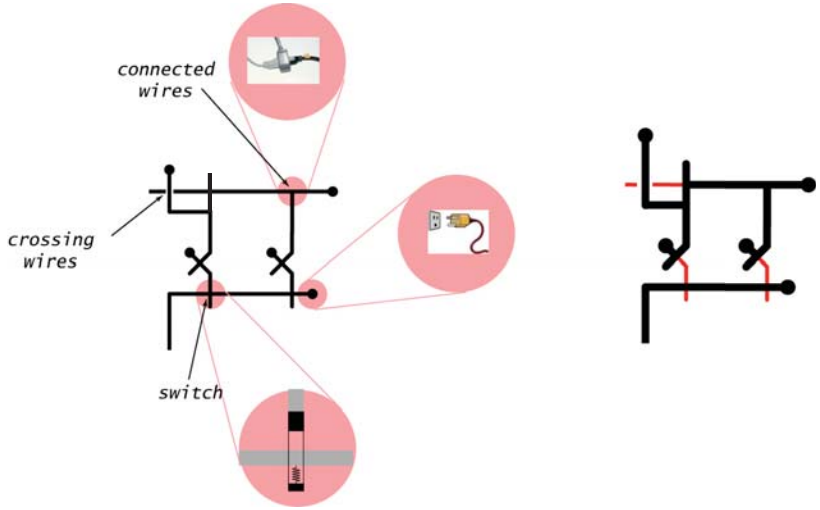
Controlled switch. [relay implementation]
3 connections: input, output, control.
Magnetic force pulls on a contact that cuts electrical flow.
Control wire affects output wire, but output does not affect control; establishes forward flow of information over time.



Anatomy of a relay (controlled switch)

Relay

Circuit Anatomy



Logic Gates: Fundamental Building Blocks

NOT = x'

x	NOT
0	1
1	0

symbol

NOT gate

OR = x+y

x y	OR
0 0	0
0 1	1
1 0	1
1 1	1

symbol

OR gate

AND = xy

x y	AND
0 0	0
0 1	0
1 0	0
1 1	1

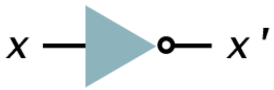
symbol

AND gate

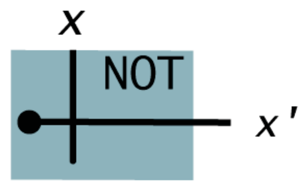
NOT

NOT = x'

x	NOT
0	1
1	0

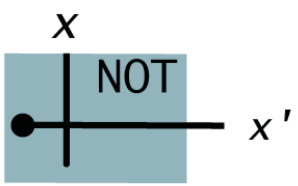


symbol



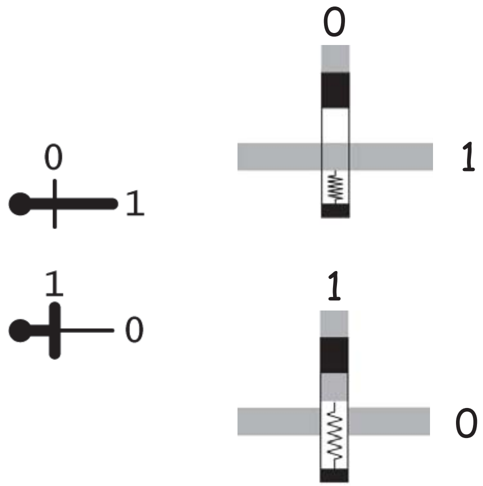
NOT gate

NOT



NOT gate

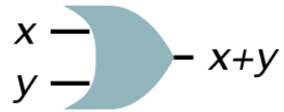
x	NOT
0	1
1	0



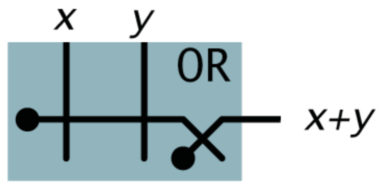
OR

$$OR = x+y$$

x	y	OR
0	0	0
0	1	1
1	0	1
1	1	1

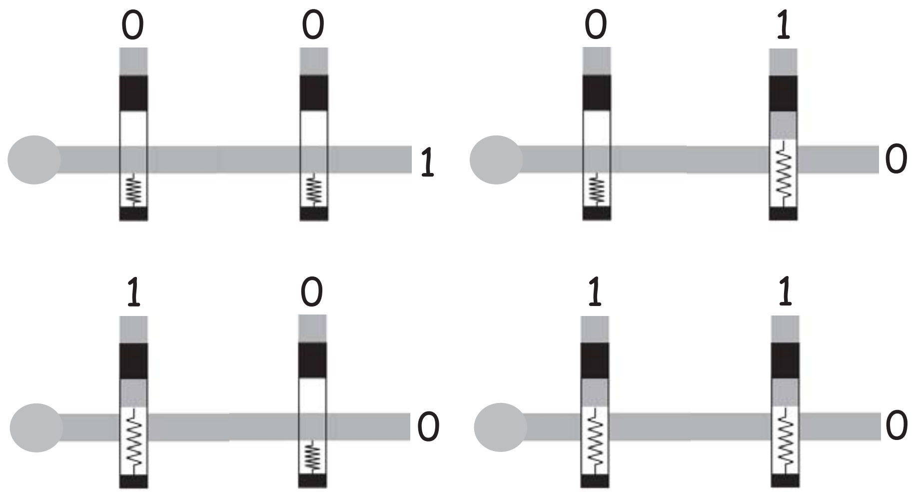


symbol

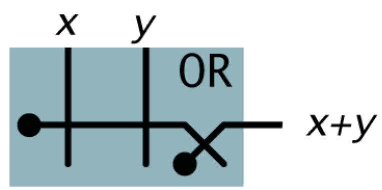


OR gate

Series relays = NOR

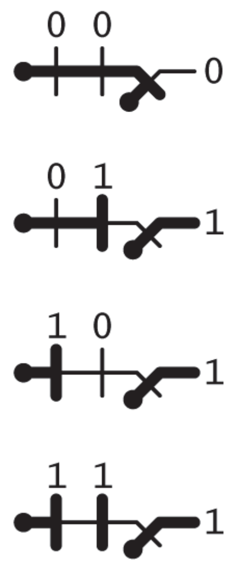


OR



OR gate

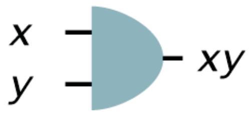
x	y	OR
0	0	0
0	1	1
1	0	1
1	1	1



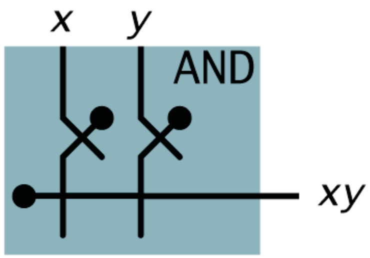
AND

AND = xy

x y	AND
0 0	0
0 1	0
1 0	0
1 1	1

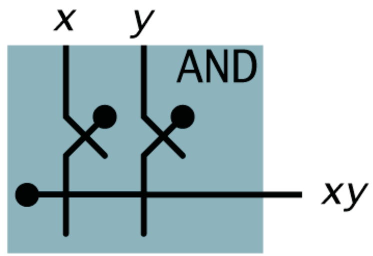


symbol



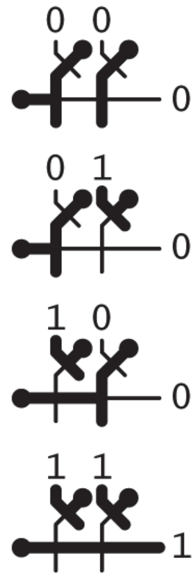
AND gate

AND



AND gate

x y	AND
0 0	0
0 1	0
1 0	0
1 1	1

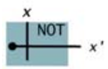


Logic Gates: Fundamental Building Blocks

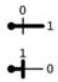
NOT = x'

x	NOT
0	1
1	0

symbol



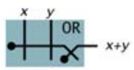
NOT gate



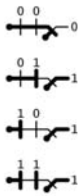
OR = x+y

x y	OR
0 0	0
0 1	1
1 0	1
1 1	1

symbol



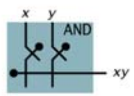
OR gate



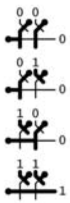
AND = xy

x y	AND
0 0	0
0 1	0
1 0	0
1 1	1

symbol

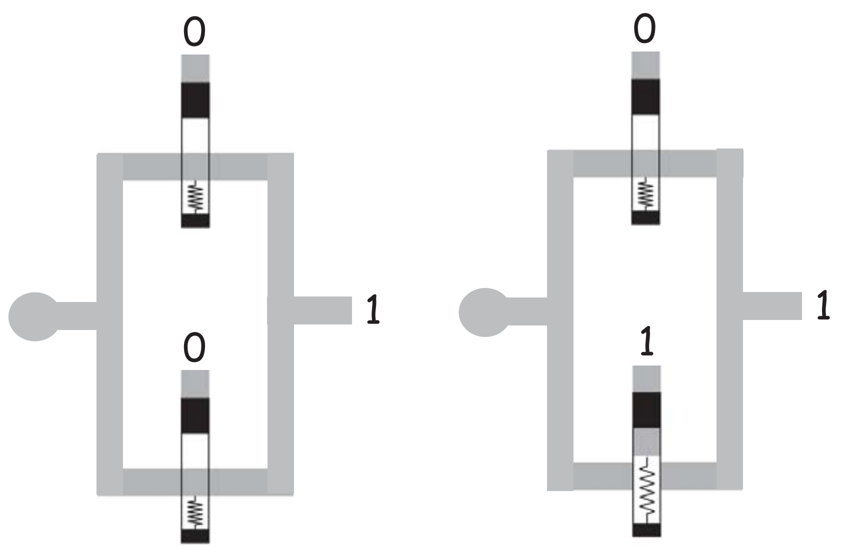


AND gate



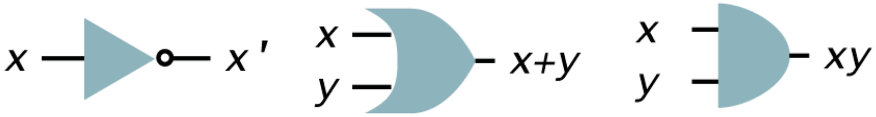
implementations with switches

What about parallel relays? =NAND



Can we implement AND/OR using parallel relays?

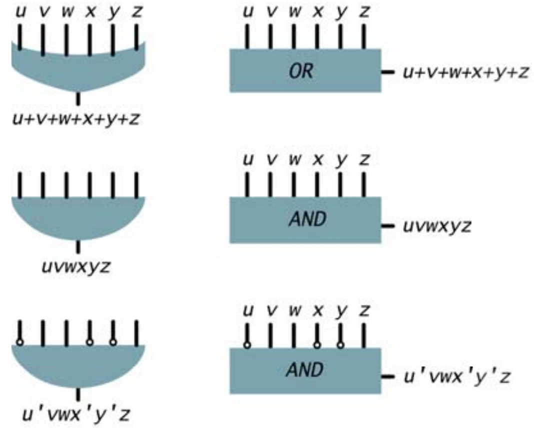
Now we know how to implement AND,OR and NOT. We can just use them as black boxes without knowing how they were implemented. Principle of information hiding.



Multiway Gates

Multiway gates.

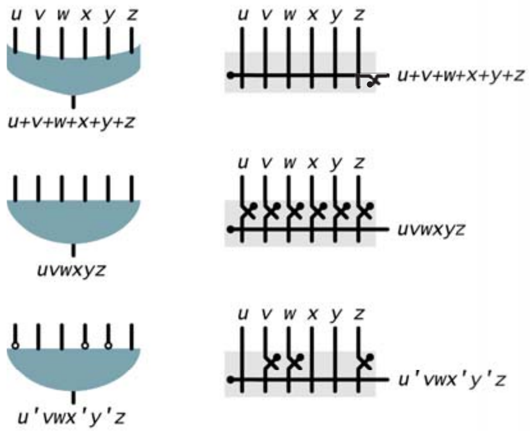
OR: 1 if any input is 1; 0 otherwise.
AND: 1 if all inputs are 1; 0 otherwise.
Generalized: negate some inputs.



Multiway Gates

Multiway gates.

OR: 1 if any input is 1; 0 otherwise.
AND: 1 if all inputs are 1; 0 otherwise.
Generalized: negate some inputs.



Multiway Gates

Multiway gates.

Can also be built from 2-way gates (less efficient but implementation independent)
Example: build 4-way OR from 2-way ORs

Translate Boolean Formula to Boolean Circuit

Sum-of-products. XOR.

XOR = x'y + xy'

x y	XOR
0 0	0
0 1	1
1 0	1
1 1	0

Truth table

Circuit

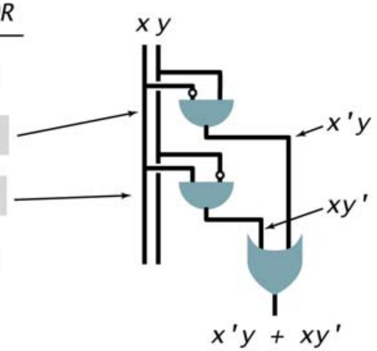
Translate Boolean Formula to Boolean Circuit

Sum-of-products. XOR.

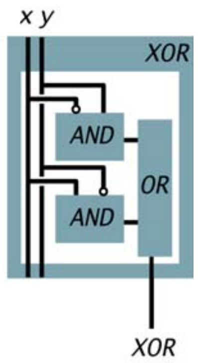
XOR = x'y + xy'

x y	XOR
0 0	0
0 1	1
1 0	1
1 1	0

Truth table



Abstract circuit



Circuit

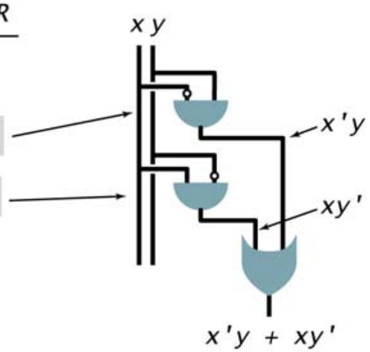
Translate Boolean Formula to Boolean Circuit

Sum-of-products. XOR.

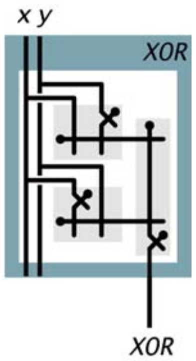
XOR = x'y + xy'

x y	XOR
0 0	0
0 1	1
1 0	1
1 1	0

Truth table



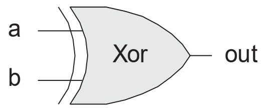
Abstract circuit



Circuit

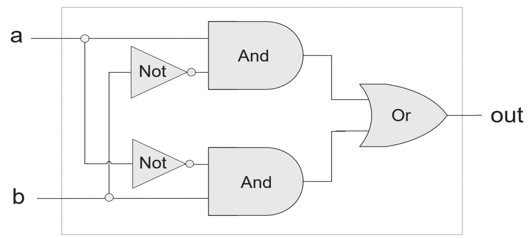
Gate logic

Interface



a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

Implementation



Xor(a,b) = Or(And(a,Not(b)),And(Not(a),b)))

Expressing a Boolean Function Using AND, OR, NOT

Ingredients.
- AND gates.
- OR gates.
- NOT gates.
- Wire.

Instructions.
- Step 1: represent input and output signals with Boolean variables.
- Step 2: construct truth table to carry out computation.
- Step 3: derive (simplified) Boolean expression using sum-of products.
- Step 4: transform Boolean expression into circuit.

Translate Boolean Formula to Boolean Circuit

Sum-of-products. Majority.

Circuit

Translate Boolean Formula to Boolean Circuit

Sum-of-products. Majority.

MAJ = x'yz + xy'z + xyz'+ xyz

x y z	MAJ
0 0 0	0
0 0 1	0
0 1 0	0
0 1 1	1
1 0 0	0
1 0 1	1
1 1 0	1
1 1 1	1

Truth table

Circuit

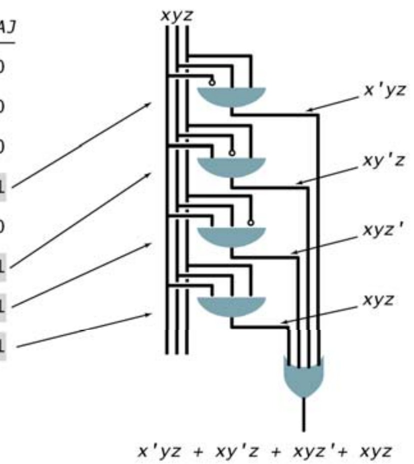
Translate Boolean Formula to Boolean Circuit

Sum-of-products. Majority.

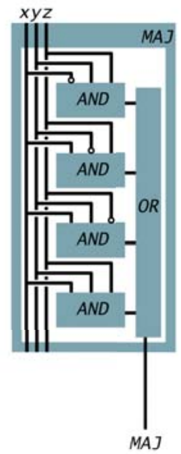
MAJ = x'yz + xy'z + xyz'+ xyz

x y z	MAJ
0 0 0	0
0 0 1	0
0 1 0	0
0 1 1	1
1 0 0	0
1 0 1	1
1 1 0	1
1 1 1	1

Truth table



Abstract circuit



Circuit

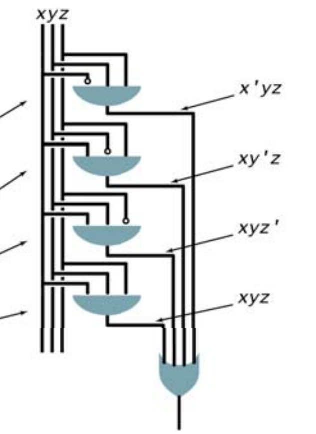
Translate Boolean Formula to Boolean Circuit

Sum-of-products. Majority.

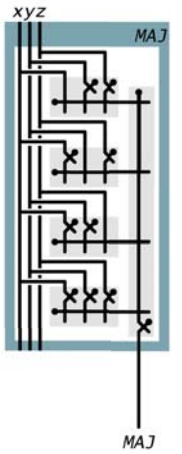
MAJ = x'yz + xy'z + xyz'+ xyz

x y z	MAJ
0 0 0	0
0 0 1	0
0 1 0	0
0 1 1	1
1 0 0	0
1 0 1	1
1 1 0	1
1 1 1	1

Truth table



Abstract circuit



Circuit

ODD Parity Circuit

ODD(x, y, z).
- 1 if odd number of inputs are 1.
- 0 otherwise.

ODD Parity Circuit

ODD(x, y, z).
- 1 if odd number of inputs are 1.
- 0 otherwise.

x	y	z	ODD	x'y'z	x'yz'	xy'z'	xyz	x'y'z + x'yz' + xy'z' + xyz
0	0	0	0	0	0	0	0	0
0	0	1	1	1	0	0	0	1
0	1	0	1	0	1	0	0	1
0	1	1	0	0	0	0	0	0
1	0	0	1	0	0	1	0	1
1	0	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0
1	1	1	1	0	0	0	1	1

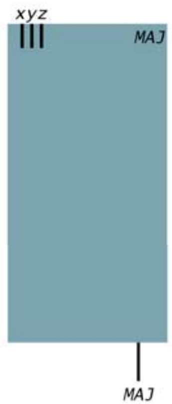
Expressing ODD using sum-of-products

ODD Parity Circuit

ODD(x, y, z).
- 1 if odd number of inputs are 1.
- 0 otherwise.

MAJ = x'yz + xy'z + xyz'+ xyz

x y z	MAJ
0 0 0	0
0 0 1	0
0 1 0	0
0 1 1	1
1 0 0	0
1 0 1	1
1 1 0	1
1 1 1	1



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

x y z	ODD
0 0 0	0
0 0 1	1
0 1 0	1
0 1 1	0
1 0 0	1
1 0 1	0
1 1 0	0
1 1 1	1

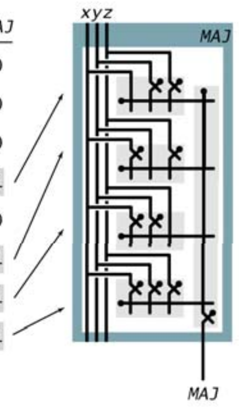
ODD Parity Circuit

ODD(x, y, z).

1 if odd number of inputs are 1.
0 otherwise.

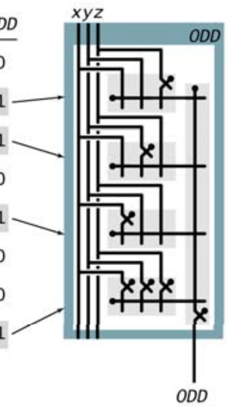
MAJ = x'yz + xy'z + xyz'+ xyz

x y z	MAJ
0 0 0	0
0 0 1	0
0 1 0	0
0 1 1	1
1 0 0	0
1 0 1	1
1 1 0	1
1 1 1	1



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

x y z	ODD
0 0 0	0
0 0 1	1
0 1 0	1
0 1 1	0
1 0 0	1
1 0 1	0
1 1 0	0
1 1 1	1



Simplification Using Boolean Algebra

Every function can be written as sum-of-product

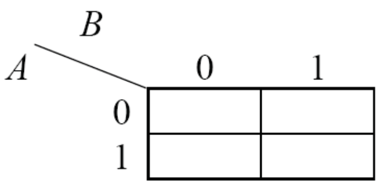
Many possible circuits for each Boolean function.

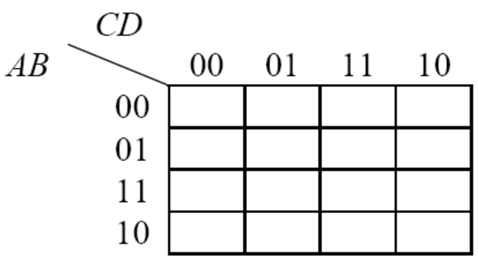
Sum-of-products not necessarily optimal in:

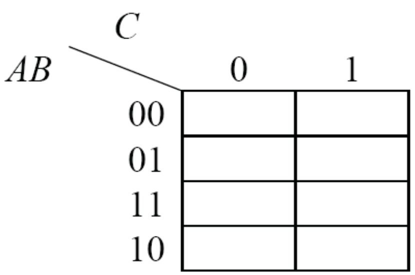
- number of switches (space)
- depth of circuit (time)

Boolean expression simplification

Karnaugh map







Karnaugh Maps (K-Maps)

- Boolean expressions can be minimized by combining terms
- K-maps minimize equations graphically
- $PA + P\bar{A} = P$

A	B	C	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

Y

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

Y

C \ AB	00	01	11	10
0	$\bar{A}\bar{B}\bar{C}$	$\bar{A}B\bar{C}$	$AB\bar{C}$	$A\bar{B}\bar{C}$
1	$\bar{A}\bar{B}C$	$\bar{A}BC$	ABC	$A\bar{B}C$

K-Map

- Circle 1's in adjacent squares
- In Boolean expression, include only literals whose true and complement form are ***not*** in the circle

A	B	C	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

Y: C \ AB	00	01	11	10
0	1	0	0	0
1	1	0	0	0

$$Y = \overline{A}\overline{B}$$

3-Input K-Map

Y: C \ AB	00	01	11	10
0	$\overline{A}\overline{B}\overline{C}$	$\overline{A}B\overline{C}$	$AB\overline{C}$	$A\overline{B}\overline{C}$
1	$\overline{A}\overline{B}C$	$\overline{A}BC$	ABC	$A\overline{B}C$

Truth Table

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

K-Map

Y: C \ AB	00	01	11	10
0				
1				

3-Input K-Map

Y: C \ AB	00	01	11	10
0	$\overline{A}\overline{B}\overline{C}$	$\overline{A}B\overline{C}$	$AB\overline{C}$	$A\overline{B}\overline{C}$
1	$\overline{A}\overline{B}C$	$\overline{A}BC$	ABC	$A\overline{B}C$

Truth Table

A	B	C	Y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

K-Map

Y: C \ AB	00	01	11	10
0	0	1	1	0
1	0	1	0	0

$$Y = \overline{A}B + B\overline{C}$$

K-Map Rules

- Every 1 must be circled at least once
- Each circle must span a power of 2 (i.e. 1, 2, 4) squares in each direction
- Each circle must be as large as possible
- A circle may wrap around the edges
- A "don't care" (X) is circled only if it helps minimize the equation

4-Input K-Map

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
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
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

Y: CD \ AB	00	01	11	10
00				
01				
11				
10				

4-Input K-Map

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
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
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

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

4-Input K-Map

A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
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
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

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

$$Y = \bar{A}C + \bar{A}BD + A\bar{B}\bar{C} + \bar{B}\bar{D}$$

4-Input K-Map with Don't care

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

Y: CD \ AB	00	01	11	10
00				
01				
11				
10				

4-Input K-Map with Don't care

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

Y

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

4-Input K-Map with Don't care

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

Y

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

$Y = A + \overline{B}\overline{D} + C$

Example

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

x y z	MAJ
0 0 0	0
0 0 1	0
0 1 0	0
0 1 1	1
1 0 0	0
1 0 1	1
1 1 0	1
1 1 1	1

xy \ z	0	1
00		
01		
11		
10		

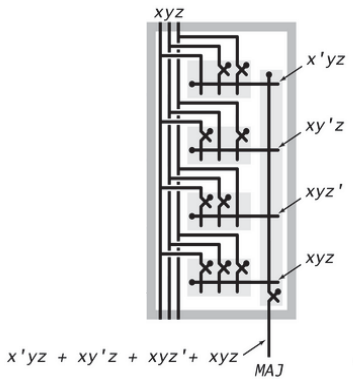
Simplification Using Boolean Algebra

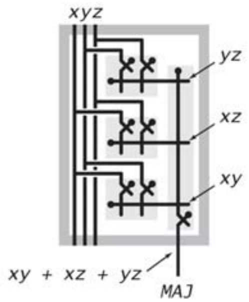
Many possible circuits for each Boolean function.

Sum-of-products not necessarily optimal in:

- number of switches (space)
- depth of circuit (time)

$MAJ(x, y, z) = x'yz + xy'z + xyz' + xyz = xy + yz + xz.$

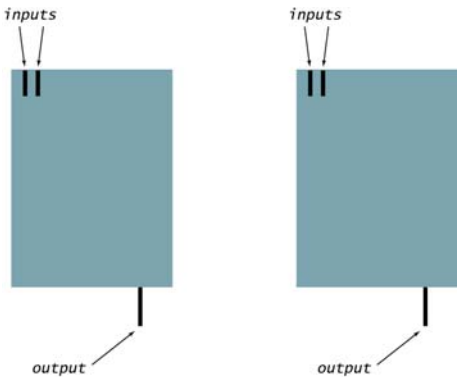




Layers of Abstraction

Layers of abstraction.

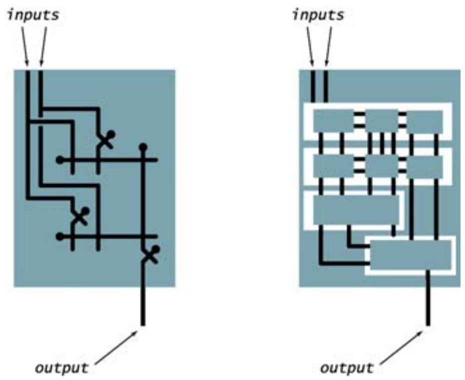
Build a circuit from wires and switches.
[implementation]
Define a circuit by its inputs and outputs. [API]
To control complexity, encapsulate circuits.
[ADT]



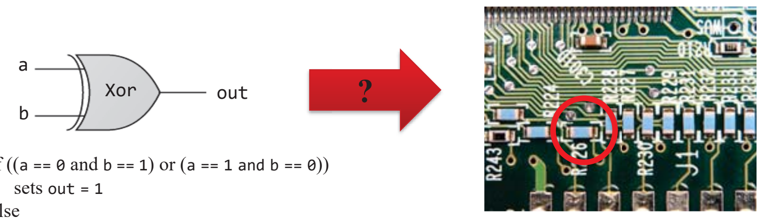
Layers of Abstraction

Layers of abstraction.

Build a circuit from wires and switches.
[implementation]
Define a circuit by its inputs and outputs. [API]
To control complexity, encapsulate circuits.
[ADT]



Building a chip

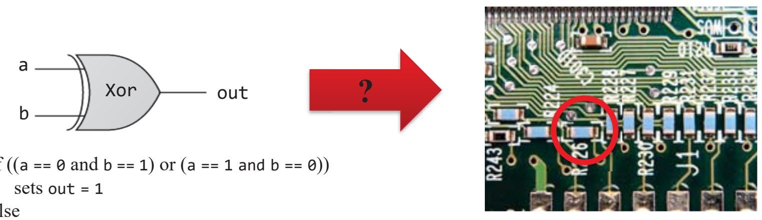


if ((a == 0 and b == 1) or (a == 1 and b == 0))
sets out = 1
else
sets out = 0

<u>The process</u>

- Design the chip architecture
- Specify the architecture in HDL
- Test the chip in a hardware simulator
- Optimize the design
- Realize the optimized design in silicon.

Building a chip



if ((a == 0 and b == 1) or (a == 1 and b == 0))
sets out = 1
else
sets out = 0

<u>The process</u>

- ✓ Design the chip architecture
- ✓ Specify the architecture in HDL
- ✓ Test the chip in a hardware simulator
- Optimize the design
- Realize the optimized design in silicon.

Chip design

```
Chip name: Xor
Inputs:    a, b
Outputs:   out
Function:  If a≠b then out=1 else out=0.
```

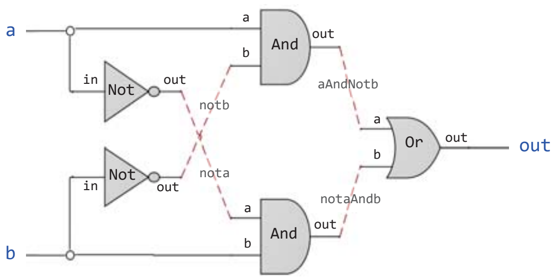
Step 1: identify input and output
Step 2: construct truth table
Step 3: derive (simplified) Boolean expression using sum-of products.
Step 4: transform Boolean expression into circuit/implement it using HDL.

You would like to test the gate before packaging.

XOR

a	b	out
0	0	
0	1	
1	0	
1	1	

Chip interfaces




```
CHIP Xor {
    IN a, b;
    OUT out;
    PARTS:
    Not (in= , out= );
    Not (in= , out= );
    And (a= , b=, out=);
    And (a= , b=, out=);
    Or  (a= , b=, out=);
}
```

If I want to use some chip-parts, how do I figure out their signatures?

Chip interfaces: Hack chip set API

Open the Hack chip set API in a window, and copy-paste chip signatures into your HDL code, as needed

```
Add16 (a= ,b= ,out= );
ALU (x= ,y= ,zx= ,nx= ,zy= ,ny= ,f= ,no= ,out= ,zr= ,ng= );
And16 (a= ,b= ,out= );
And (a= ,b= ,out= );
Aregister (in= ,load= ,out= );
Bit (in= ,load= ,out= );
CPU (inM= ,instruction= ,reset= ,outM= ,writeM= ,ad
DFF (in= ,out= );
DMux4Way (in= ,sel= ,a= ,b= ,c= ,d= );
DMux8Way (in= ,sel= ,a= ,b= ,c= ,d= ,e= ,f= ,g= ,h=
Dmux (in= ,sel= ,a= ,b= );
Dregister (in= ,load= ,out= );
FullAdder (a= ,b= ,c= ,sum= ,carry= );
HalfAdder (a= ,b= ,sum= , carry= );
Inc16 (in= ,out= );
Keyboard (out= );
Memory (in= ,load= ,address= ,out= );
Mux16 (a= ,b= ,sel= ,out= );
Mux4Way16 (a= ,b= ,c= ,d= ,sel= ,out= );
Mux8Way16 (a= ,b= ,c= ,d= ,e= ,f= ,g= ,h= ,sel= ,ou
```

```
Mux8Way (a= ,b= ,c= ,d= ,e= ,f= ,g= ,h= ,sel= ,out= );
Mux (a= ,b= ,sel= ,out= );
Nand (a= ,b= ,out= );
Not16 (in= ,out= );
Not (in= ,out= );
Or16 (a= ,b= ,out= );
Or8Way (in= ,out= );
Or (a= ,b= ,out= );
PC (in= ,load= ,inc= ,reset= ,out= );
PCLoadLogic (cinstr= ,j1= ,j2= ,j3= ,load= ,inc= );
RAM16K (in= ,load= ,address= ,out= );
RAM4K (in= ,load= ,address= ,out= );
RAM512 (in= ,load= ,address= ,out= );
RAM64 (in= ,load= ,address= ,out= );
RAM8 (in= ,load= ,address= ,out= );
Register (in= ,load= ,out= );
ROM32K (address= ,out= );
Screen (in= ,load= ,address= ,out= );
Xor (a= ,b= ,out= );
```

Built-in chips

```
CHIP Foo {
    IN ...;
    OUT ...;

    PARTS:
    ...
    Bar(...)
    ...
}
```

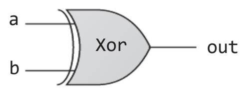
Q: Suppose you want to use a chip-part before you've implemented it. How to do it?

A: The simulator features built-in implementations of all the project 1 chips

Forcing the simulator to use a built-in chip, say `Bar`:

- Typically, `Bar.hdl` will be either a given stub-file, or a file that has an incomplete implementation
- Remove, or rename, the file `Bar.hdl` from the project folder
- Whenever `Bar` will be mentioned as a chip-part in some chip definition, the simulator will fail to find `Bar.hdl` in the current folder. This will cause the simulator to invoke the built-in version of `Bar` instead.

Design: Requirements



a	b	out
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

if ((a == 0 and b == 1) or (a == 1 and b == 0))
    sets out = 1
else
    sets out = 1

Requirement

Build a chip that delivers this functionality

```
/** Sets out = (a And Not(b)) Or (Not(a) And b)) */
CHIP Xor {
    IN a, b;
    OUT out;
    PARTS:
    // Missing implementation
```

Gate Interface

Expressed as an HDL *stub file*

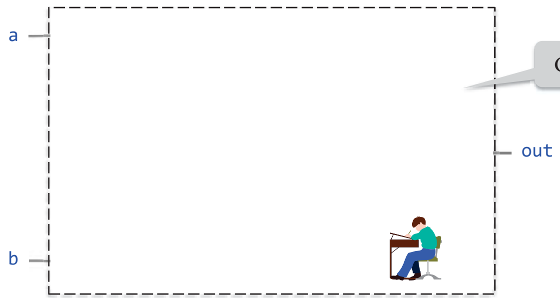
Design: Implementation

```
/** Sets out = (a And Not(b)) Or (Not(a) And b)) */
CHIP Xor {
    IN a, b;
    OUT out;
    PARTS:
    // Missing implementation
```

Gate Interface

Expressed as an HDL *stub file*

Design: Implementation



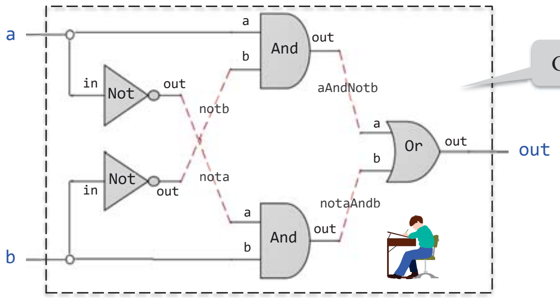
Gate diagram

```
/** Sets out = (a And Not(b)) Or (Not(a) And b)) */
CHIP Xor {
    IN a, b;
    OUT out;
    PARTS:
    // Missing implementation
```

Gate Interface

Expressed as an HDL *stub file*

Design: Implementation



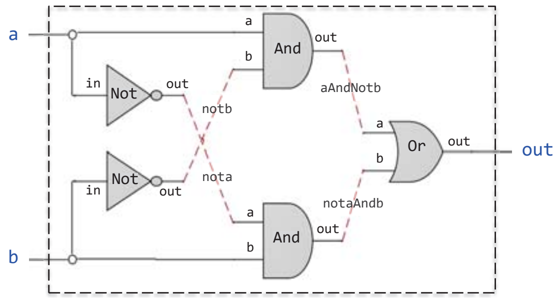
Gate diagram

```
/** Sets out = (a And Not(b)) Or (Not(a) And b)) */
CHIP Xor {
    IN a, b;
    OUT out;
    PARTS:
    // Missing implementation
```

<u>Gate Interface</u>

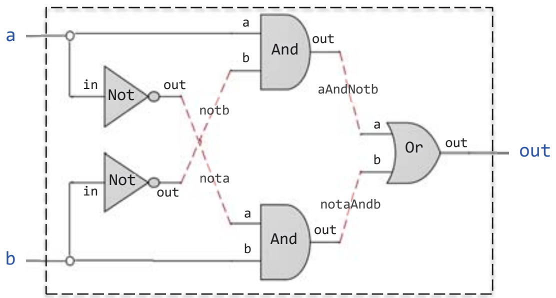
Expressed as an HDL *stub file*

Design: Implementation



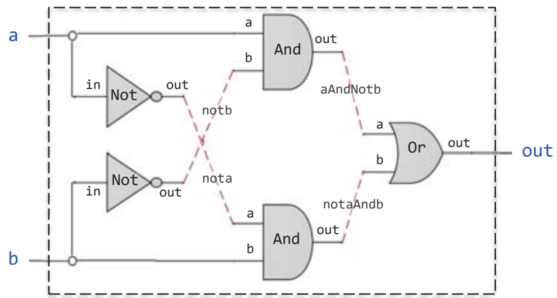

```
/** Sets out = (a And Not(b)) Or (Not(a) And b)) */
CHIP Xor {
    IN a, b;
    OUT out;
    PARTS:
    // Missing implementation
```

Design: Implementation




```
/** Sets out = (a And Not(b)) Or (Not(a) And b)) */
CHIP Xor {
    IN a, b;
    OUT out;
    PARTS:
    Not (in=a, out=nota);
    Not (in=b, out=notb);
    And (a=a, b=notb, out=aAndNotb);
    And (a=nota, b=b, out=notaAndb);

}
```

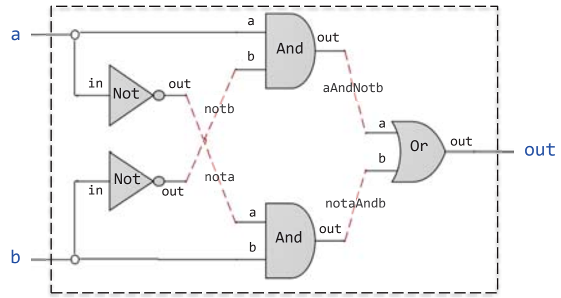
Design: Implementation




```
/** Sets out = (a And Not(b)) Or (Not(a) And b)) */
CHIP Xor {
    IN a, b;
    OUT out;
    PARTS:
    Not (in=a, out=nota);
    Not (in=b, out=notb);
    And (a=a, b=notb, out=aAndNotb);
    And (a=nota, b=b, out=notaAndb);

}
```

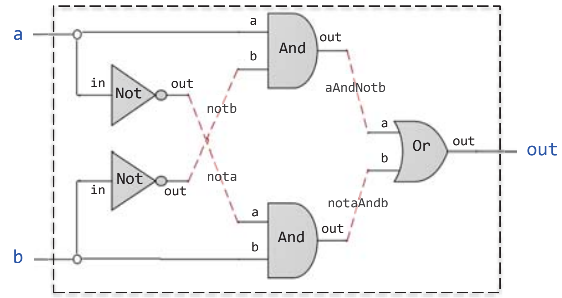
<u>Practice:</u> Complete the missing HDL code

Design: Implementation




```
/** Sets out = (a And Not(b)) Or (Not(a) And b)) */
CHIP Xor {
    IN a, b;
    OUT out;
    PARTS:
    Not (in=a, out=nota);
    Not (in=b, out=notb);
    And (a=a, b=notb, out=aAndNotb);
    And (a=nota, b=b, out=notaAndb);
    Or  (a=aAndNotb, b=notaAndb, out=out);
}
```

Design: Implementation



gate interface

gate Implement -ation


```
/** Sets out = (a And Not(b)) Or (Not(a) And b)) */
CHIP Xor {
    IN a, b;
    OUT out;
    PARTS:
    Not (in=a, out=nota);
    Not (in=b, out=notb);
    And (a=a, b=notb, out=aAndNotb);
    And (a=nota, b=b, out=notaAndb);
    Or  (a=aAndNotb, b=notaAndb, out=out);
}
```


<u>A logic gate has:</u>

- One interface
- Many possible implementations

Hardware description languages

<u>Observations:</u>

- HDL is a functional / declarative language
- An HDL program can be viewed as a textual specification of a chip diagram
- The order of HDL statements is insignificant.


```
/** Sets out = (a And Not(b)) Or (Not(a) And b)) */
CHIP Xor {
    IN a, b;
    OUT out;
    PARTS:
    Not (in=a, out=nota);
    Not (in=b, out=notb);
    And (a=a, b=notb, out=aAndNotb);
    And (a=nota, b=b, out=notaAndb);
    Or  (a=aAndNotb, b=notaAndb, out=out);
}
```


Hardware description languages

<u>Common HDLs</u>

- VHDL
- Verilog
- …

<u>Our HDL</u>

- Similar in spirit to other HDLs
- Minimal and simple
- Provides all you need for this course

Our HDL Guide / Documentation:

The Elements of Computing Systems / Appendix 2: HDL


```
/** Sets out = (a And Not(b)) Or (Not(a) And b)) */
CHIP Xor {
    IN a, b;
    OUT out;
    PARTS:
    Not (in=a, out=nota);
    Not (in=b, out=notb);
    And (a=a, b=notb, out=aAndNotb);
    And (a=nota, b=b, out=notaAndb);
    Or  (a=aAndNotb, b=notaAndb, out=out);
}
```

Multi-bit bus

- Sometimes we wish to manipulate a *sequence of bits* as a single entity
- Such a multi-bit entity is termed "bus"

<u>Example: 16-bit bus</u>

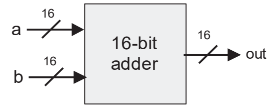
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	0	0	0	1	1	0	1	1	1	0	1

MSB = Most significant bit

LSB = Least significant bit

Working with buses: Example

```
/* Adds two 16-bit values. */
CHIP Adder {
   IN a[16], b[16];
   OUT out[16];

   PARTS:
   ...
}
```

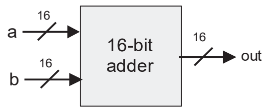


	15	…	1	0
a:	1	…	1	1
b:	0	…	1	0
c:	0	…	0	1
out:	1	…	1	0

```
/* Adds three 16-bit inputs. */
CHIP Adder3Way {
   IN a[16], b[16], c[16];
   OUT out[16];
```

Working with buses: Example

```
/* Adds two 16-bit values. */
CHIP Adder {
   IN a[16], b[16];
   OUT out[16];

   PARTS:
   ...
}
```

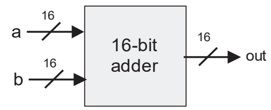


	15	…	1	0
a:	1	…	1	1
b:	0	…	1	0
c:	0	…	0	1
out:	1	…	1	0

```
/* Adds three 16-bit inputs. */
CHIP Adder3Way {
   IN a[16], b[16], c[16];
   OUT out[16];

   PARTS:
   Adder(a=  , b= , out=   );
   Adder(a=  , b= , out=   );
}
```

Working with buses: Example

```
/* Adds two 16-bit values. */
CHIP Adder {
   IN a[16], b[16];
   OUT out[16];

   PARTS:
   ...
}
```



	15	…	1	0
a:	1	…	1	1
b:	0	…	1	0
c:	0	…	0	1
out:	1	…	1	0

```
/* Adds three 16-bit inputs. */
CHIP Adder3Way {
   IN a[16], b[16], c[16];
   OUT out[16];

   PARTS:
   Adder(a=a , b=b, out=ab);
   Adder(a=ab, b=c, out=out);
}
```

n-bit value (bus) can be treated as a single entity

Creates an internal bus pin (ab)

Working with individual bits within buses

```
/* Returns 1 if a==1 and b==1,
 * 0 otherwise. */
CHIP And {
   IN a, b;
   OUT out;
   ...
}
```

```
    3  2  1  0
a: [0][1][1][1]

      out: [0]
```

```
/* 4-way And: Ands 4 bits. */
CHIP And4Way {
   IN a[4];
   OUT out;
```

Working with individual bits within buses

```
/* Returns 1 if a==1 and b==1,
 * 0 otherwise. */
CHIP And {
   IN a, b;
   OUT out;
   ...
}
```

```
    3  2  1  0
a: [0][1][1][1]

      out: [0]
```

```
/* 4-way And: Ands 4 bits. */
CHIP And4Way {
   IN a[4];
   OUT out;

   PARTS:
   And(a=        , b=     , out=        );
   And(a=        , b=     , out=        );
   And(a=        , b=     , out=        );
}
```

Working with individual bits within buses

```
/* Returns 1 if a==1 and b==1,
 * 0 otherwise. */
CHIP And {
   IN a, b;
   OUT out;
   ...
}
```

```
    3  2  1  0
a: [0][1][1][1]

      out: [0]
```

```
/* 4-way And: Ands 4 bits. */
CHIP And4Way {
   IN a[4];
   OUT out;

   PARTS:
   And(a=a[0],    b=a[1], out=and01);
   And(a=and01,   b=a[2], out=and012);
   And(a=and012,  b=a[3], out=out);
}
```

Input bus pins can be subscripted.

Working with individual bits within buses

```
/* Returns 1 if a==1 and b==1,
 * 0 otherwise. */
CHIP And {
   IN a, b;
   OUT out;
   ...
}
```

```
    3  2  1  0
a: [0][1][1][1]

      out: [0]
```

```
/* 4-way And: Ands 4 bits. */
CHIP And4Way {
   IN a[4];
   OUT out;

   PARTS:
   And(a=a[0],    b=a[1], out=and01);
   And(a=and01,   b=a[2], out=and012);
   And(a=and012,  b=a[3], out=out);
}
```

Input bus pins can be subscripted.

```
      3  2  1  0
a:   [0][1][0][1]

b:   [0][0][1][1]

out: [0][0][0][1]
```

```
/* Bit-wise And of two 4-bit inputs */
CHIP And4 {
   IN a[4], b[4];
   OUT out[4];
```

Working with individual bits within buses

```
/* Returns 1 if a==1 and b==1,
 * 0 otherwise. */
CHIP And {
   IN a, b;
   OUT out;
   ...
}
```

	3	2	1	0
a:	0	1	1	1
out:				0

```
/* 4-way And: Ands 4 bits. */
CHIP And4Way {
   IN a[4];
   OUT out;

   PARTS:
   And(a=a[0],   b=a[1], out=and01);
   And(a=and01,  b=a[2], out=and012);
   And(a=and012, b=a[3], out=out);
}
```

Input bus pins can be subscripted.

	3	2	1	0
a:	0	1	0	1
b:	0	0	1	1
out:	0	0	0	1

```
/* Bit-wise And of two 4-bit inputs */
CHIP And4 {
   IN a[4], b[4];
   OUT out[4];

   PARTS:
   And(a=    , b=    , out=      );
   And(a=    , b=    , out=      );
   And(a=    , b=    , out=.     );
   And(a=    , b=    , out=      );
}
```

Working with individual bits within buses

```
/* Returns 1 if a==1 and b==1,
 * 0 otherwise. */
CHIP And {
   IN a, b;
   OUT out;
   ...
}
```

	3	2	1	0
a:	0	1	1	1
out:				0

```
/* 4-way And: Ands 4 bits. */
CHIP And4Way {
   IN a[4];
   OUT out;

   PARTS:
   And(a=a[0],   b=a[1], out=and01);
   And(a=and01,  b=a[2], out=and012);
   And(a=and012, b=a[3], out=out);
}
```

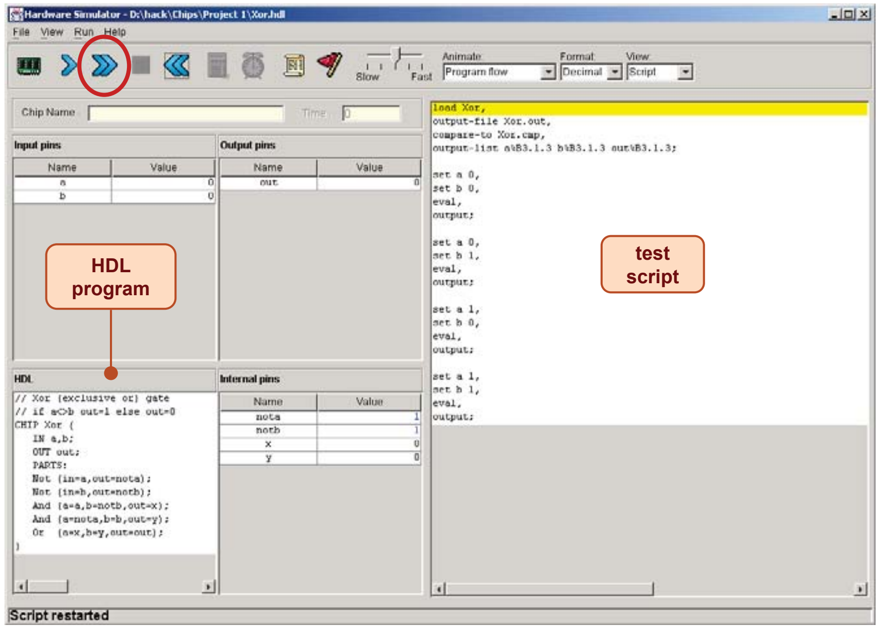
Input bus pins can be subscripted.

	3	2	1	0
a:	0	1	0	1
b:	0	0	1	1
out:	0	0	0	1

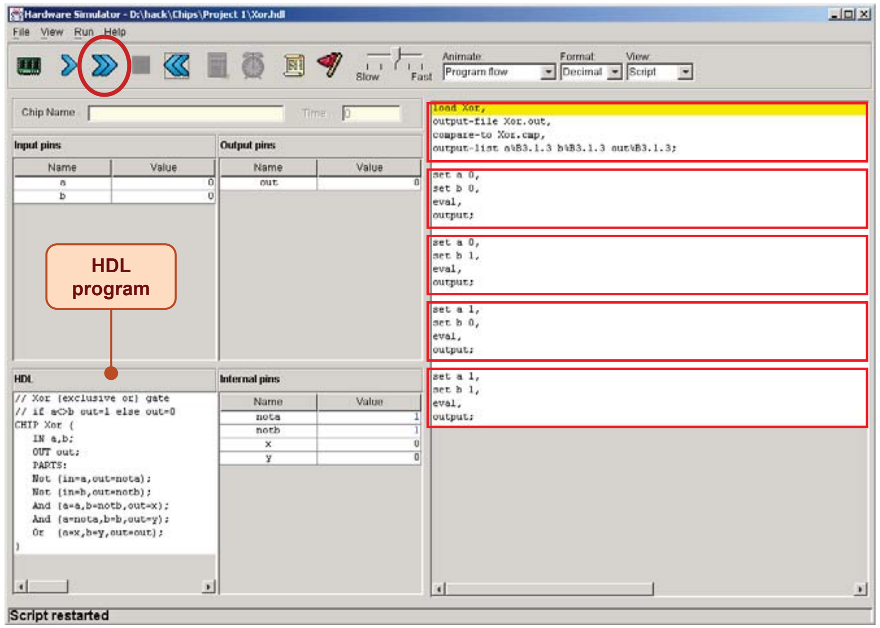
```
/* Bit-wise And of two 4-bit inputs */
CHIP And4 {
   IN a[4], b[4];
   OUT out[4];

   PARTS:
   And(a=a[0], b=b[0], out=out[0]);
   And(a=a[1], b=b[1], out=out[1]);
   And(a=a[2], b=b[2], out=out[2]);
   And(a=a[3], b=b[3], out=out[3]);
}
```

Output bus pins can be subscripted

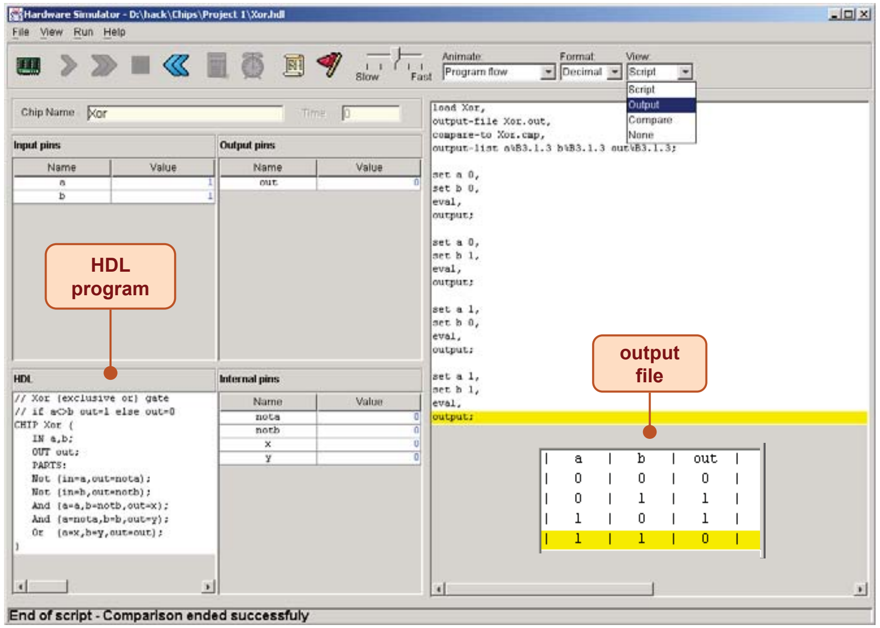
Hardware simulator (demonstrating Xor gate construction)



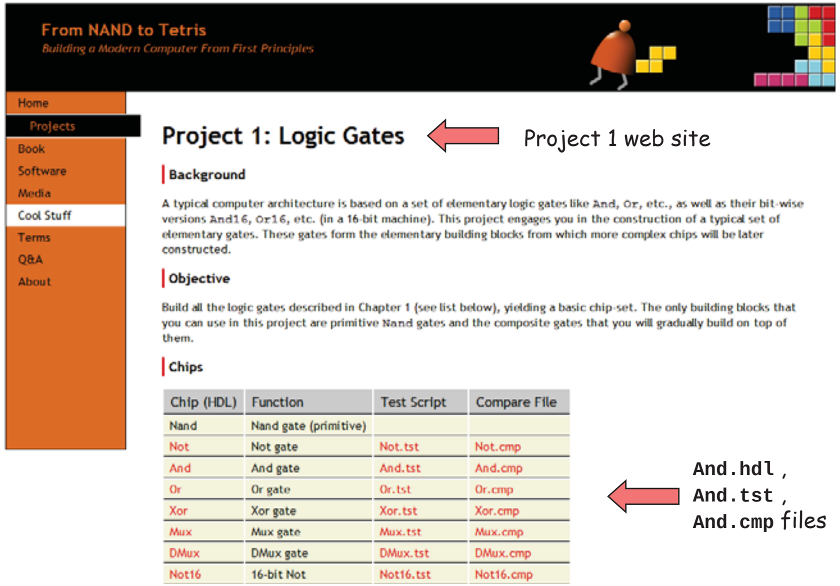
Hardware simulator



Hardware simulator



Project materials: www.nand2tetris.org



Project 1 tips

- Read the Introduction + Chapter 1 of the book
- Download the book's software suite
- Go through the hardware simulator tutorial
- Do Project 0 (optional)
- You're in business.

Gates for project #1 (Basic Gates)

```
Chip name: Not
Inputs:    in
Outputs:   out
Function:  If in=0 then out=1 else out=0.
```

```
Chip name: And
Inputs:    a, b
Outputs:   out
Function:  If a=b=1 then out=1 else out=0.
```

```
Chip name: Or
Inputs:    a, b
Outputs:   out
Function:  If a=b=0 then out=0 else out=1.
```

```
Chip name: Xor
Inputs:    a, b
Outputs:   out
Function:  If a≠b then out=1 else out=0.
```

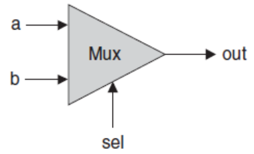
Gates for project #1

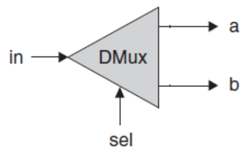
```
Chip name: Mux
Inputs:    a, b, sel
Outputs:   out
Function:  If sel=0 then out=a else out=b.
```

```
Chip name: DMux
Inputs:    in, sel
Outputs:   a, b
Function:  If sel=0 then {a=in, b=0} else {a=0, b=in}.
```

sel	out
0	a
1	b

sel	a	b
0	in	0
1	0	in





Gates for project #1 (Multi-bit version)

```
Chip name: Not16
Inputs:    in[16] // a 16-bit pin
Outputs:   out[16]
Function:  For i=0..15 out[i]=Not(in[i]).
```

```
Chip name: And16
Inputs:    a[16], b[16]
Outputs:   out[16]
Function:  For i=0..15 out[i]=And(a[i],b[i]).
```

```
Chip name: Or16
Inputs:    a[16], b[16]
Outputs:   out[16]
Function:  For i=0..15 out[i]=Or(a[i],b[i]).
```

```
Chip name: Mux16
Inputs:    a[16], b[16], sel
Outputs:   out[16]
Function:  If sel=0 then for i=0..15 out[i]=a[i]
           else for i=0..15 out[i]=b[i].
```

Gates for project #1 (Multi-way version)

```
Chip name: Or8Way
Inputs:    in[8]
Outputs:   out
Function:  out=Or(in[0],in[1],...,in[7]).
```

Gates for project #1 (Multi-way version)

sel[1]	sel[0]	out
0	0	a
0	1	b
1	0	c
1	1	d

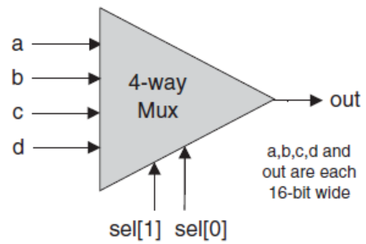


Figure 1.10 4-way multiplexor. The width of the input and output buses may vary.

sel[1]	sel[0]	a	b	c	d
0	0	in	0	0	0
0	1	0	in	0	0
1	0	0	0	in	0
1	1	0	0	0	in

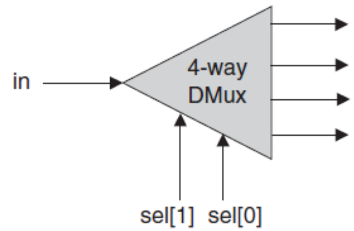


Figure 1.11 4-way demultiplexor.

Gates for project #1 (Multi-way version)

```
Chip name: Mux4Way16
Inputs:    a[16], b[16], c[16], d[16], sel[2]
Outputs:   out[16]
Function:  If sel=00 then out=a else if sel=01 then out=b
           else if sel=10 then out=c else if sel=11 then out=d
Comment:   The assignment operations mentioned above are all
           16-bit. For example, "out=a" means "for i=0..15
           out[i]=a[i]".
```

```
Chip name: Mux8Way16
Inputs:    a[16],b[16],c[16],d[16],e[16],f[16],g[16],h[16],
           sel[3]
Outputs:   out[16]
Function:  If sel=000 then out=a else if sel=001 then out=b
           else if sel=010 out=c ... else if sel=111 then out=h
Comment:   The assignment operations mentioned above are all
           16-bit. For example, "out=a" means "for i=0..15
           out[i]=a[i]".
```



```
Chip name: DMux4Way
Inputs:    in, sel[2]
Outputs:   a, b, c, d
Function:  If sel=00 then      {a=in, b=c=d=0}
           else if sel=01 then {b=in, a=c=d=0}
           else if sel=10 then {c=in, a=b=d=0}
           else if sel=11 then {d=in, a=b=c=0}.
```

```
Chip name: DMux8Way
Inputs:    in, sel[3]
Outputs:   a, b, c, d, e, f, g, h
Function:  If sel=000 then      {a=in, b=c=d=e=f=g=h=0}
           else if sel=001 then {b=in, a=c=d=e=f=g=h=0}
           else if sel=010 ...
           ...
           else if sel=111 then {h=in, a=b=c=d=e=f=g=0}.
```