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:

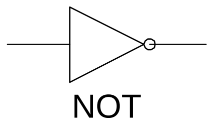
- NOT, AND, OR

NOT

$\overline{X}$ X'

X	¬X
F	T
T	F

Digital gate diagram for NOT:

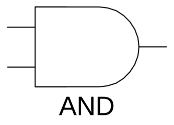


AND

$X{\cdot}Y$ XY

X	Y	X ∧ Y
F	F	F
F	T	F
T	F	F
T	T	T

Digital gate diagram for AND:

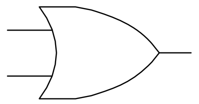


OR

X+Y

X	**Y**	**X ∨ Y**
F	F	F
F	T	T
T	F	T
T	T	T

Digital gate diagram for OR:



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 Equations

Example 1:

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

Simplifying Boolean Equations

Example 1:

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

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

  $= B(1)$

  $= B$

Simplifying Boolean Equations

Example 2:

- $Y = A(AB + ABC)$

Simplifying Boolean Equations

Example 2:

- $Y = A(AB + ABC)$

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

  $= A(AB(1))$

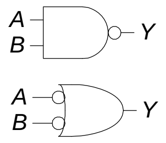
  $= A(AB)$

  $= (AA)B$

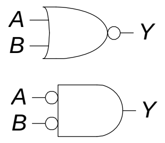
  $= AB$

DeMorgan's Theorem

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

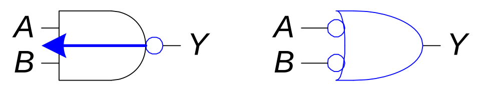


- $Y = \overline{A + B} = \overline{A} \bullet \overline{B}$

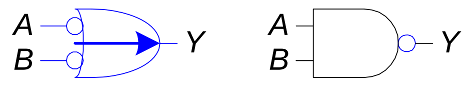


Bubble Pushing

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

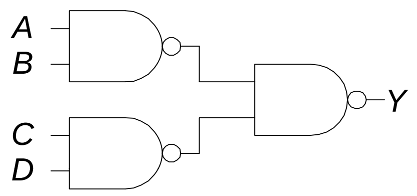


- **Forward:**
  - Body changes
  - Adds bubble to output



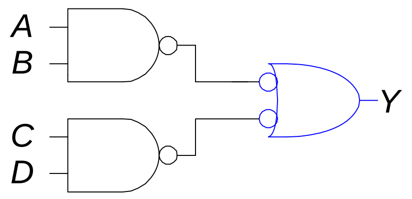
Bubble Pushing

- What is the Boolean expression for this circuit?



Bubble Pushing

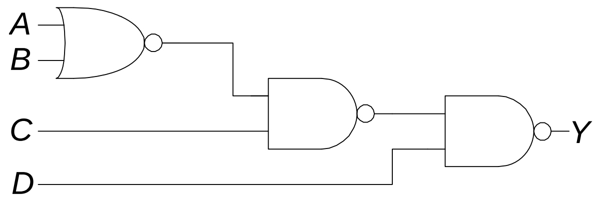
- What is the Boolean expression for this circuit?



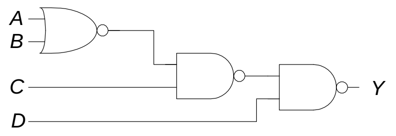
$$Y = AB + CD$$

Bubble Pushing Rules

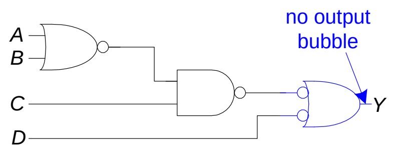
- Begin at output, then work toward inputs
- Push bubbles on final output back
- Draw gates in a form so bubbles cancel



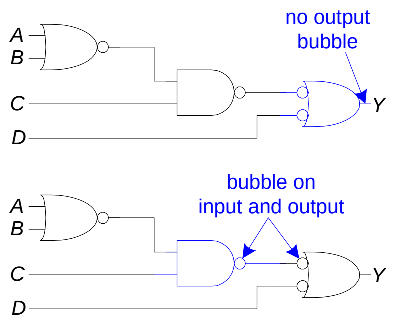
Bubble Pushing Example



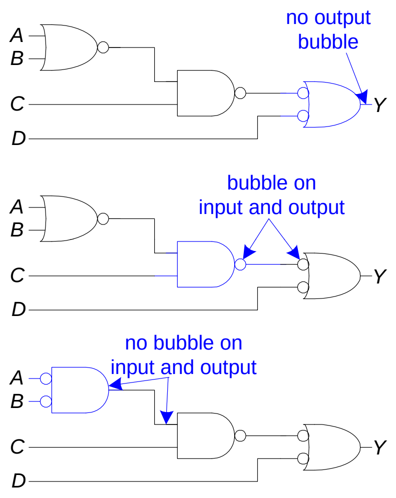
Bubble Pushing Example



Bubble Pushing Example



Bubble Pushing Example



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

Truth Tables (1 of 3)

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	$\neg$X	Y	$\neg$X $\vee$ Y
F	T	F	T
F	T	T	T
T	F	F	F
T	F	T	T

Truth Tables (2 of 3)

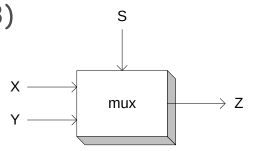
Example: $X \wedge \neg Y$

X	Y	$\neg$Y	X $\wedge$ $\neg$Y
F	F	T	F
F	T	F	F
T	F	T	T
T	T	F	F

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

X	Y	S	Y $\wedge$ S	$\neg$S	X $\wedge$ $\neg$S	(Y $\wedge$ S) $\vee$ (X $\wedge$ $\neg$S)
F	F	F	F	T	F	F
F	T	F	F	T	F	F
T	F	F	F	T	T	T
T	T	F	F	T	T	T
F	F	T	F	F	F	F
F	T	T	T	F	F	T
T	F	T	F	F	F	F
T	T	T	T	F	F	T

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	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

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

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

From Math to Real-World implementation

We can implement any Boolean function using NAND gates only.

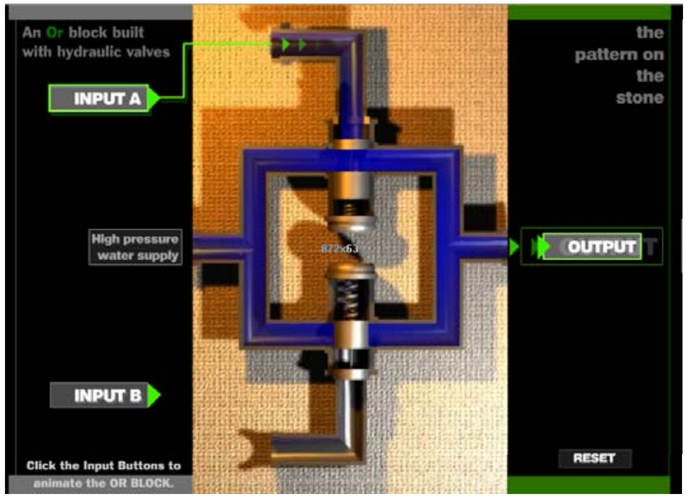
We talk about abstract Boolean algebra (logic) so far.

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

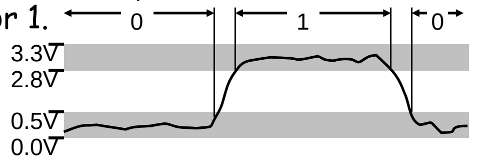
Fluid switch (http://www.cs.princeton.edu/introcs/lectures/fluid-computer.swf


An Or block built with hydraulic valves
INPUT A
High pressure water supply
INPUT B
Click the Input Buttons to animate the OR BLOCK.
the pattern on the stone
OUTPUT
RESET

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.

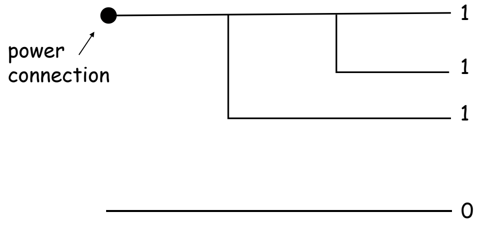
Basic abstractions.

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

Wires

Wires.

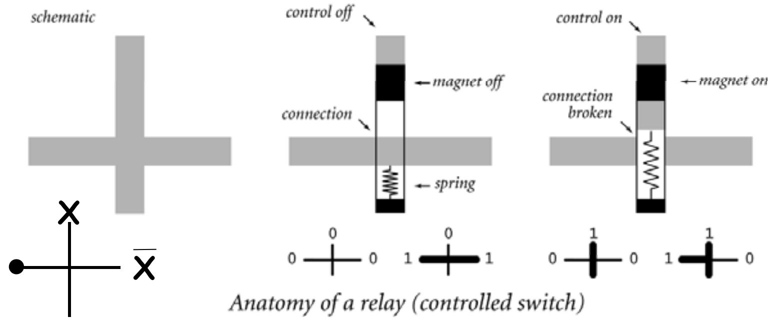
- On (1): connected to power.
- Off (0): not connected to power.
- If a wire is connected to a wire that is on, that wire is also on.
- Typical drawing convention: "flow" from top, left to bottom, right.



Controlled Switch

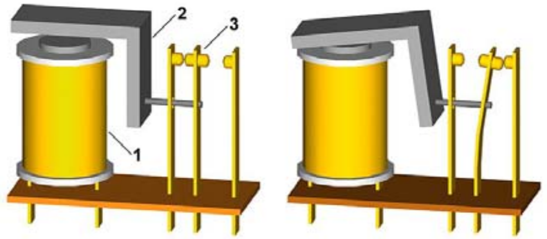
Controlled switch. [relay implementation]

- 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.

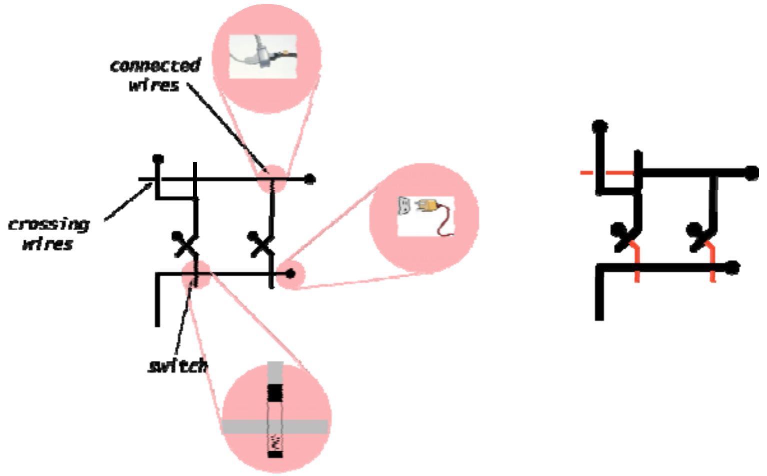


Anatomy of a relay (controlled switch)

Relay



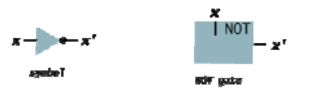
Circuit Anatomy



Logic Gates: Fundamental Building Blocks

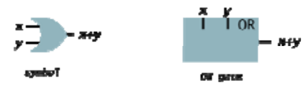
NOT = x'

x	NOT
0	1
1	0



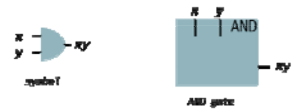
OR = x+y

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



AND = xy

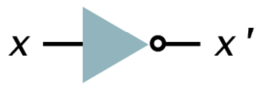
x y	AND
0 0	0
0 1	0
1 0	0
1 1	1



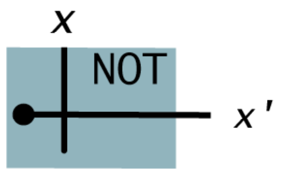
NOT

NOT = x'

x	NOT
0	1
1	0

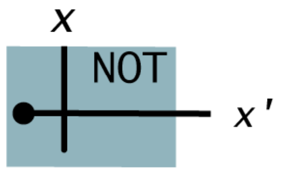


symbol



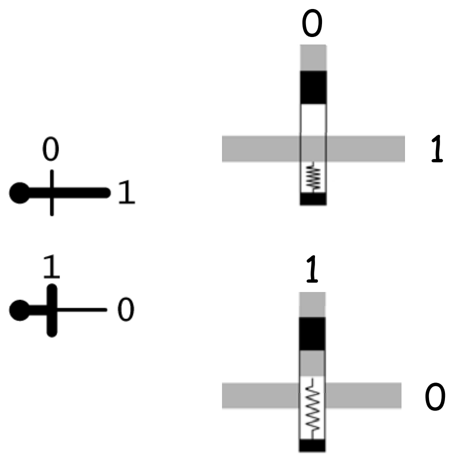
NOT gate

NOT



NOT gate

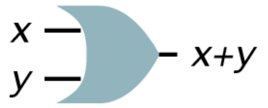
x	NOT
0	1
1	0



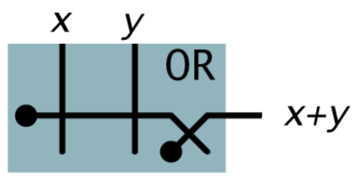
OR

OR = x+y

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

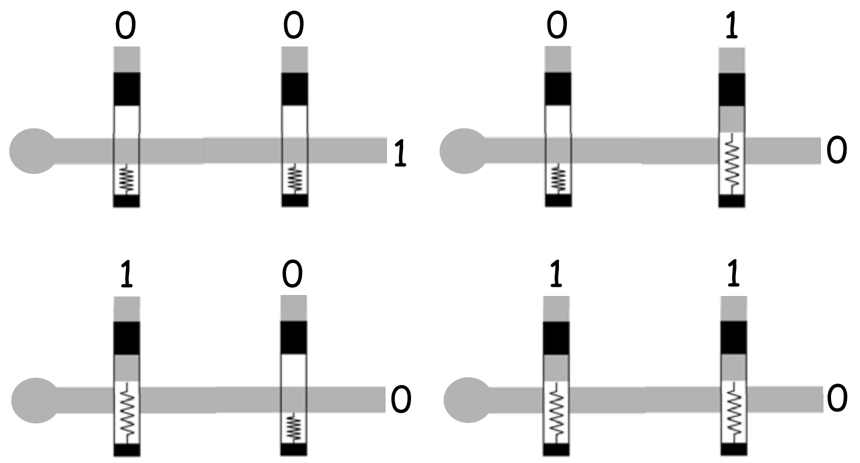


symbol

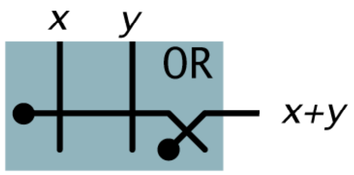


OR gate

Series relays = NOR

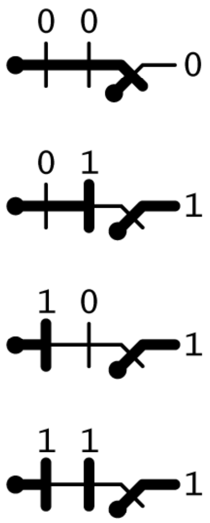


OR



OR gate

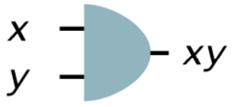
x y	OR
0 0	0
0 1	1
1 0	1
1 1	1



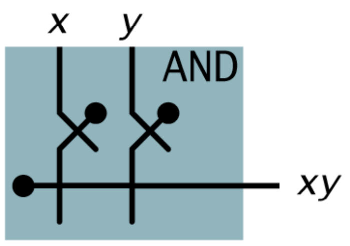
AND

AND = xy

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

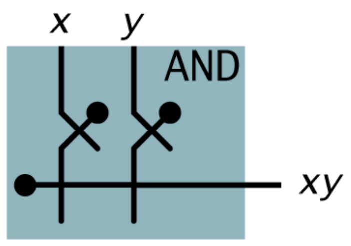


symbol



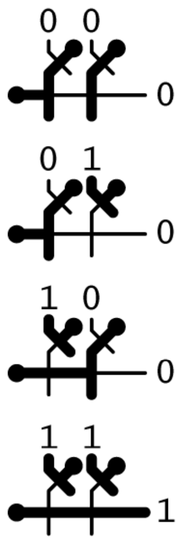
AND gate

AND



AND gate

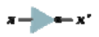
x y	AND
0 0	0
0 1	0
1 0	0
1 1	1



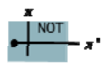
Logic Gates: Fundamental Building Blocks

NOT = x'

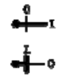
x	NOT
0	1
1	0



symbol

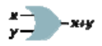


NOT gate

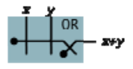


OR = x+y

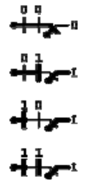
x y	OR
0 0	0
0 1	1
1 0	1
1 1	1



symbol

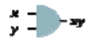


OR gate

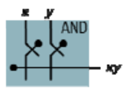


AND = xy

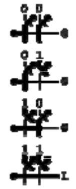
x y	AND
0 0	0
0 1	0
1 0	0
1 1	1



symbol

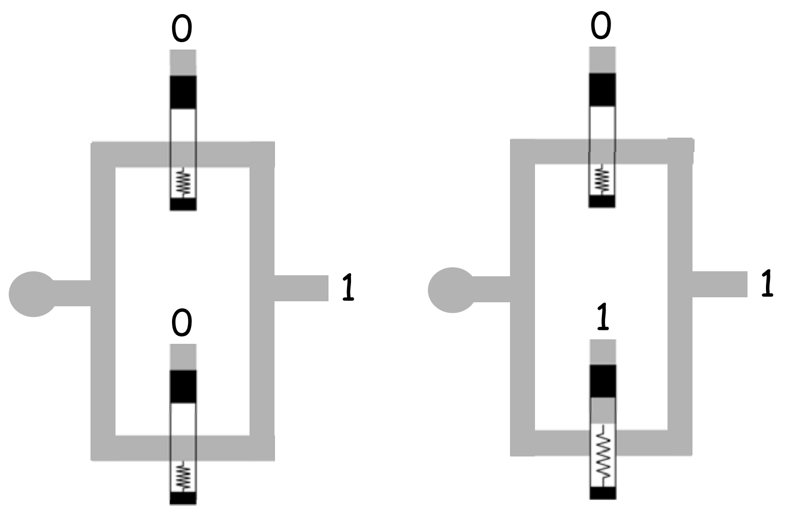


AND gate



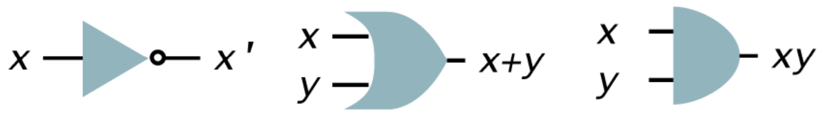
implementations with switches

What about parallel relays? =NAND



Can we implement AND/OR using parallel relays?

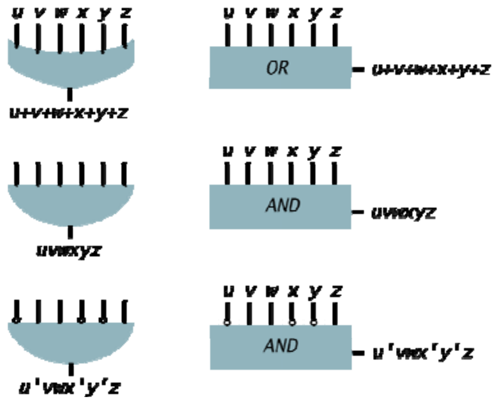
Now we know how to implement AND,OR and NOT. We can just use them as black boxes without knowing how they were implemented. Principle of information hiding.



Multiway Gates

Multiway gates.

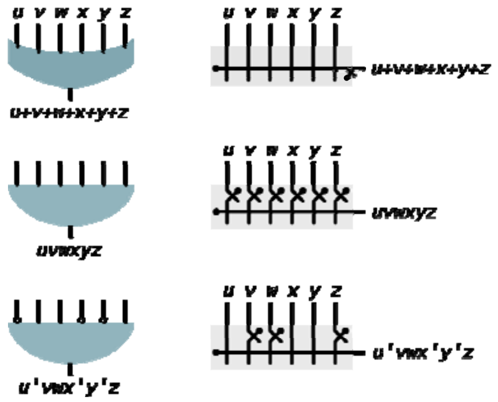
- OR: 1 if any input is 1; 0 otherwise.
- AND: 1 if all inputs are 1; 0 otherwise.
- Generalized: negate some inputs.



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

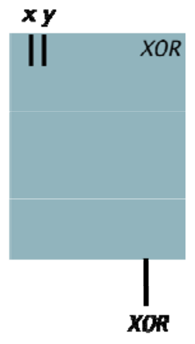
Translate Boolean Formula to Boolean Circuit

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

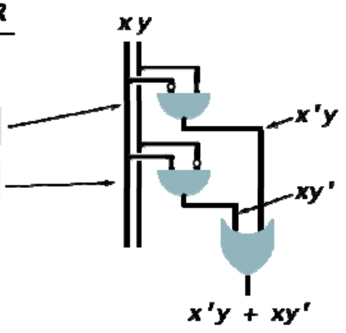
Translate Boolean Formula to Boolean Circuit

Sum-of-products. XOR.

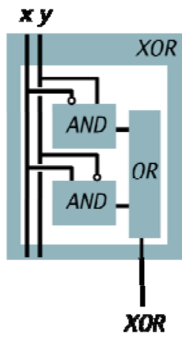
$XOR = x'y + xy'$

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

Truth table



Abstract circuit



Circuit

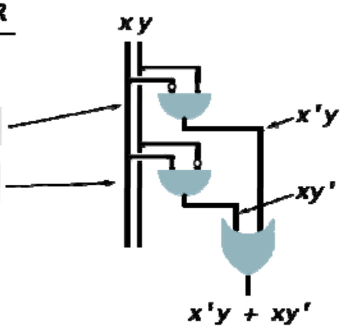
Translate Boolean Formula to Boolean Circuit

Sum-of-products. XOR.

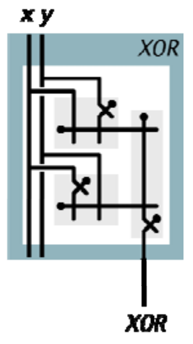
$XOR = x'y + xy'$

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

Truth table



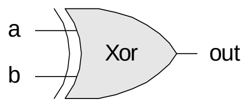
Abstract circuit



Circuit

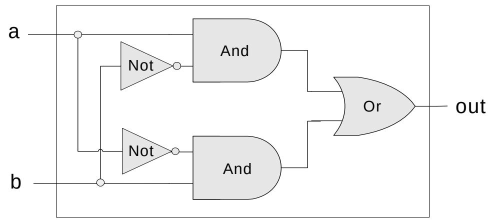
Gate logic

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)))

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

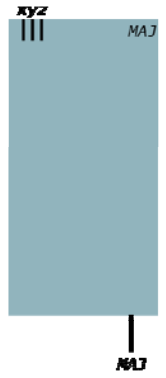
ODD Parity Circuit

ODD(x, y, z).

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

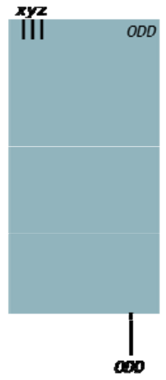
MAJ = x'yz + xy'z + xyz'+ xyz

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



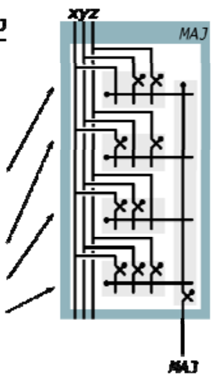
ODD Parity Circuit

ODD(x, y, z).

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

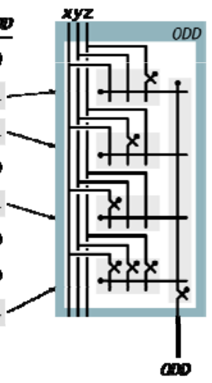
MAJ = x'yz + xy'z + xyz'+ xyz

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



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

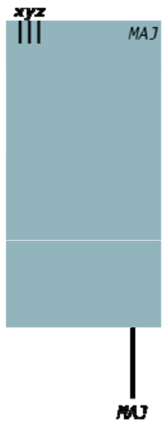
Translate Boolean Formula to Boolean Circuit

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

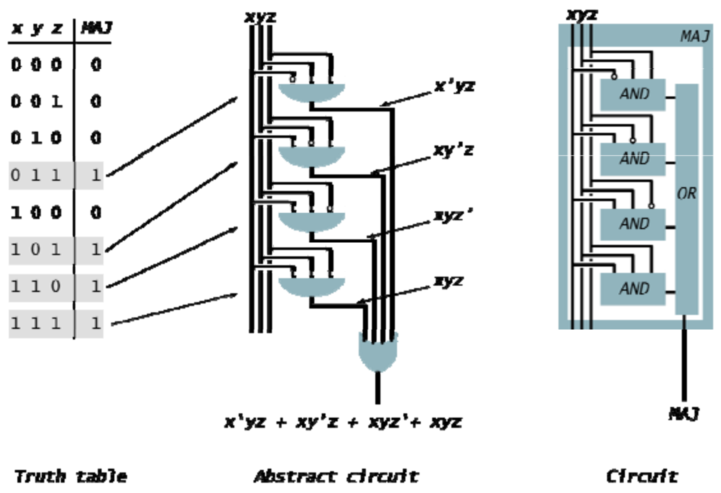


Circuit

Translate Boolean Formula to Boolean Circuit

Sum-of-products. Majority.

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

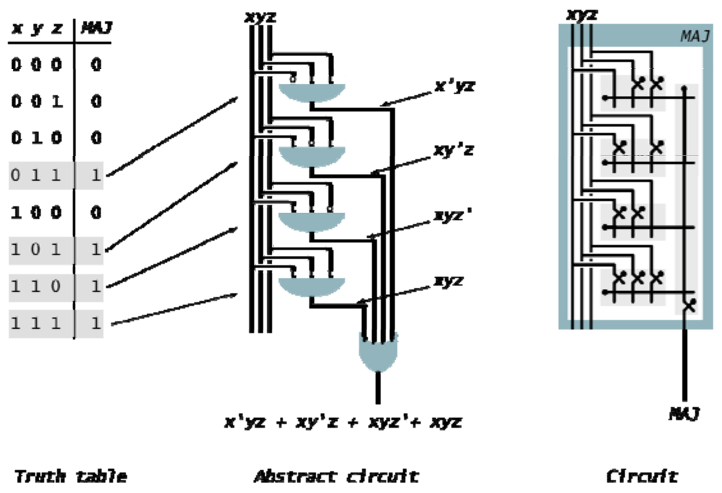


x y z	MAJ
0 0 0	0
0 0 1	0
0 1 0	0
0 1 1	1
1 0 0	0
1 0 1	1
1 1 0	1
1 1 1	1

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

Simplification Using Boolean Algebra

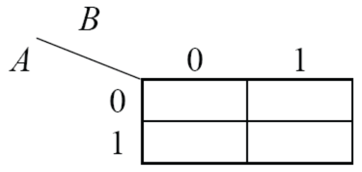
Every function can be written as sum-of-product

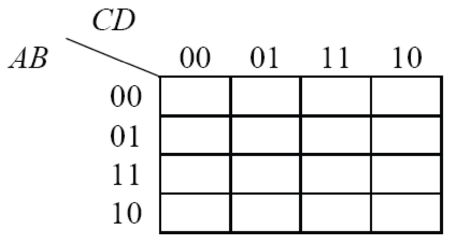
Many possible circuits for each Boolean function.

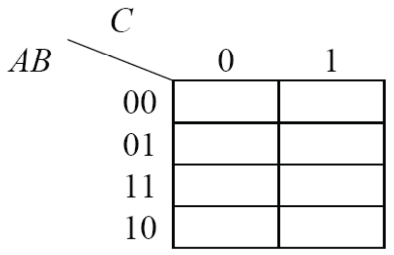
- Sum-of-products not necessarily optimal in:
  - number of switches (space)
  - depth of circuit (time)

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

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

3-Input K-Map

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	$A\bar{B}C$

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	0
1	1	1	1

K-Map

Y: C \ AB	00	01	11	10
0				
1				

3-Input K-Map

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	$A\bar{B}C$

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	0
1	1	1	1

K-Map

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

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

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

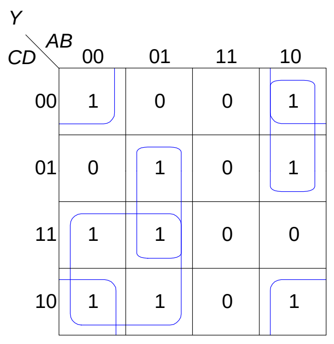
A	*B*	*C*	*D*	*Y*
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

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

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

K-Maps with Don't Cares

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				

K-Maps with Don't Cares

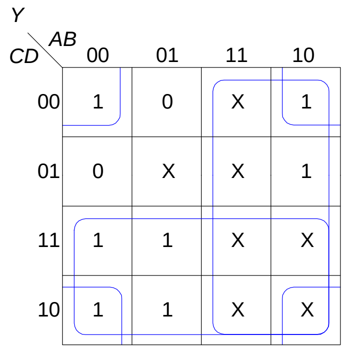
A	B	C	D	Y
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	X
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

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

K-Maps with Don't Cares

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

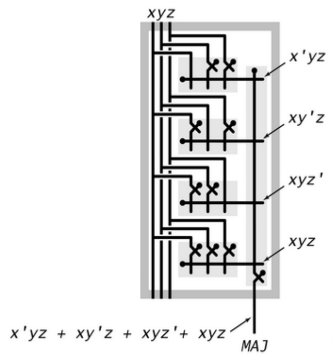
xy \ z	0	1
00		
01		
11		
10		

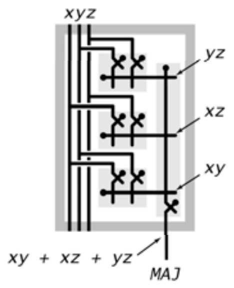
Simplification Using Boolean Algebra

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.

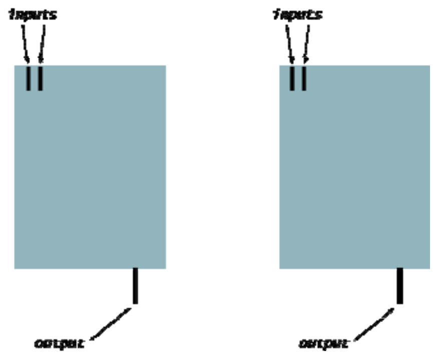




Layers of Abstraction

Layers of abstraction.

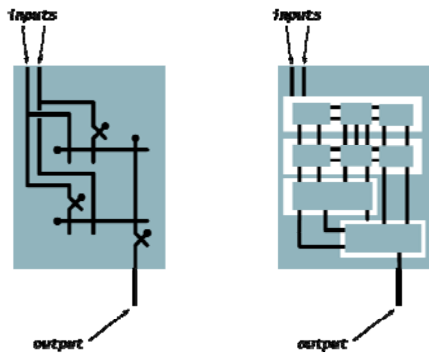
- Build a circuit from wires and switches. [implementation]
- Define a circuit by its inputs and outputs. [API]
- To control complexity, encapsulate circuits. [ADT]



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]



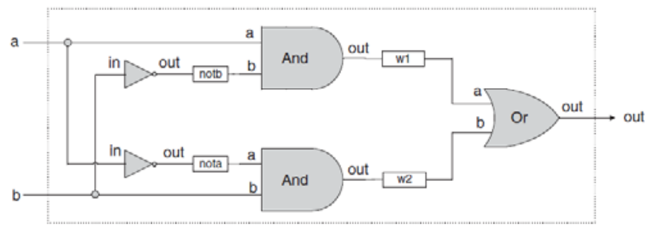
Specification

```
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.

HDL



HDL program (Xor.hdl)

```
/* Xor (exclusive or) gate:
   If a<>b out=1 else out=0. */
CHIP Xor {
   IN a, b;
   OUT out;
   PARTS:
   Not(in=a, out=nota);
   Not(in=b, out=notb);
   And(a=a, b=notb, out=w1);
   And(a=nota, b=b, out=w2);
   Or(a=w1, b=w2, out=out);
}
```

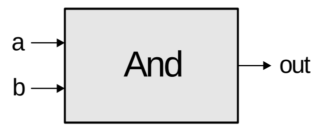
Test script (Xor.tst)

```
load Xor.hdl,
output-list a, b, out;
set a 0, set b 0,
eval, output;
set a 0, set b 1,
eval, output;
set a 1, set b 0,
eval, output;
set a 1, set b 1,
eval, output;
```

Output file (Xor.out)

```
a | b | out
- - - - - - -
0 | 0 | 0
0 | 1 | 1
1 | 0 | 1
1 | 1 | 0
```

Example: Building an **And** gate



And.cmp

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

Contract:

When running your **.hdl** on our **.tst**, your **.out** should be the same as our **.cmp**.

And.hdl

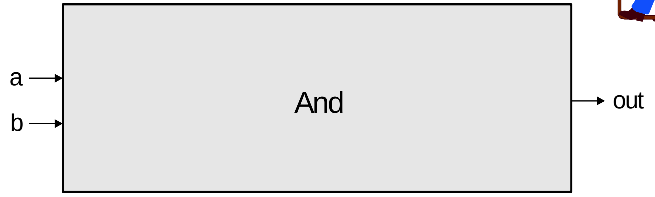
```
CHIP And
{   IN  a, b;
    OUT out;
    // implementation missing
}
```

And.tst

```
load And.hdl,
output-file And.out,
compare-to And.cmp,
output-list a b out;
set a 0,set b 0,eval,output;
set a 0,set b 1,eval,output;
set a 1,set b 0,eval,output;
set a 1, set b 1, eval, output;
```

Building an **And** gate

Interface: And(a,b) = 1 exactly when a=b=1

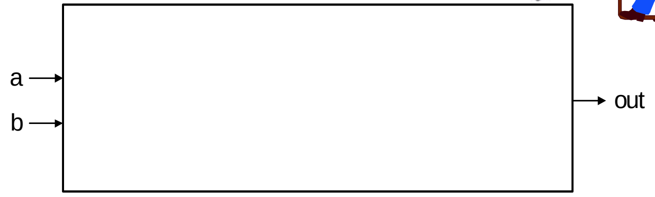


And.hdl

```
CHIP And
{   IN  a, b;
    OUT out;
    // implementation missing
}
```

Building an **And** gate

Implementation: And(a,b) = Not(Nand(a,b))

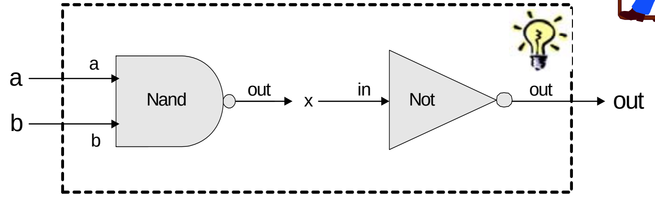


And.hdl

```
CHIP And
{   IN  a, b;
    OUT out;
    // implementation missing
}
```

Building an **And** gate

Implementation: And(a,b) = Not(Nand(a,b))

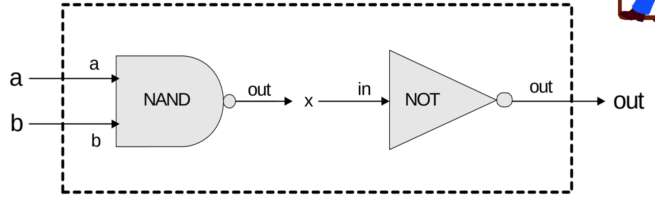


And.hdl

```
CHIP And
{   IN  a, b;
    OUT out;
    // implementation missing
}
```

Building an **And** gate

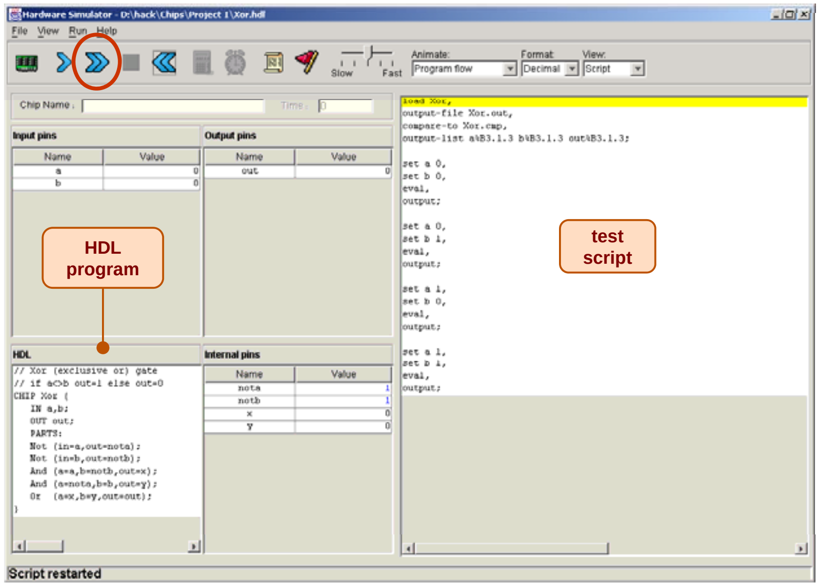
Implementation: And(a,b) = Not(Nand(a,b))



And.hdl

```
CHIP And
{   IN  a, b;
    OUT out;
    Nand(a = a,
         b = b,
         out = x);
    Not(in = x, out = out)
}
```

Hardware simulator (demonstrating Xor gate construction)



Input pins

Name	Value
a	0
b	0

Output pins

Name	Value
out	0

```
load Xor,
output-file Xor.out,
compare-to Xor.cmp,
output-list a%B3.1.3 b%B3.1.3 out%B3.1.3;

set a 0,
set b 0,
eval,
output;

set a 0,
set b 1,
eval,
output;

set a 1,
set b 0,
eval,
output;

set a 1,
set b 1,
eval,
output;
```

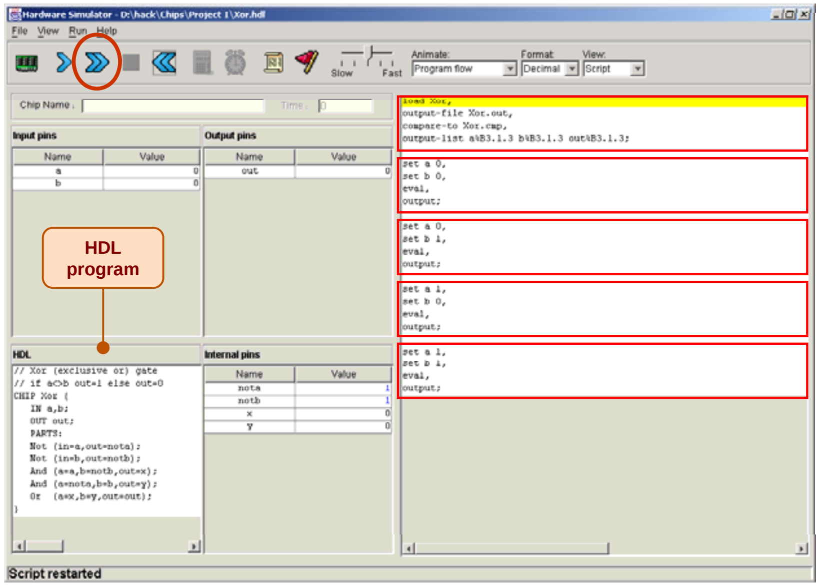
```
// Xor (exclusive or) gate
// if a<>b out=1 else out=0
CHIP Xor {
   IN a,b;
   OUT out;
   PARTS:
   Not (in=a,out=nota);
   Not (in=b,out=notb);
   And (a=a,b=notb,out=x);
   And (a=nota,b=b,out=y);
   Or  (a=x,b=y,out=out);
}
```

Internal pins

Name	Value
nota	1
notb	1
x	0
y	0

Script restarted

Hardware simulator



Input pins

Name	Value
a	0
b	0

Output pins

Name	Value
out	0

```
load Xor,
output-file Xor.out,
compare-to Xor.cmp,
output-list a%B3.1.3 b%B3.1.3 out%B3.1.3;

set a 0,
set b 0,
eval,
output;

set a 0,
set b 1,
eval,
output;

set a 1,
set b 0,
eval,
output;

set a 1,
set b 1,
eval,
output;
```

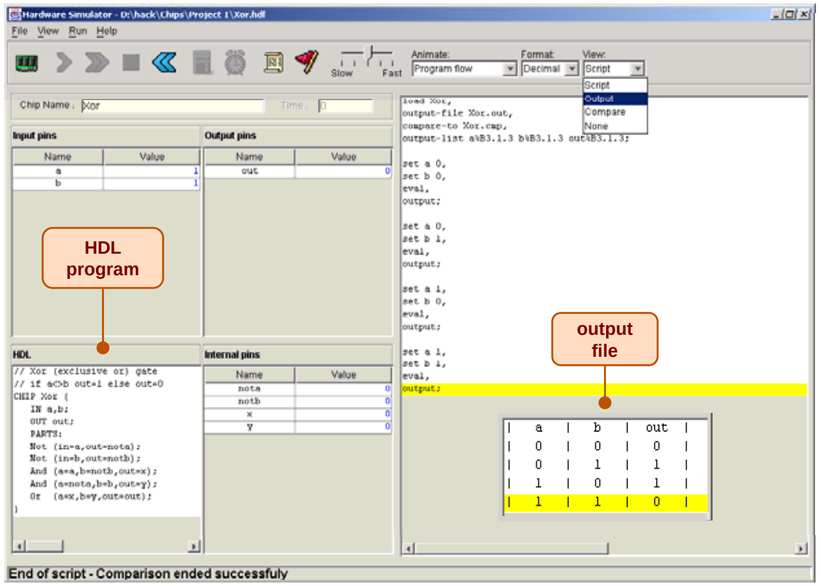
```
// Xor (exclusive or) gate
// if a<>b out=1 else out=0
CHIP Xor {
   IN a,b;
   OUT out;
   PARTS:
   Not (in=a,out=nota);
   Not (in=b,out=notb);
   And (a=a,b=notb,out=x);
   And (a=nota,b=b,out=y);
   Or  (a=x,b=y,out=out);
}
```

Internal pins

Name	Value
nota	1
notb	1
x	0
y	0

Script restarted

Hardware simulator



Name	Value
a	1
b	1

Name	Value
out	0

```
// Xor (exclusive or) gate
// if a<>b out=1 else out=0
CHIP Xor {
   IN a,b;
   OUT out;
   PARTS:
   Not (in=a,out=nota);
   Not (in=b,out=notb);
   And (a=a,b=notb,out=x);
   And (a=nota,b=b,out=y);
   Or  (a=x,b=y,out=out);
}
```

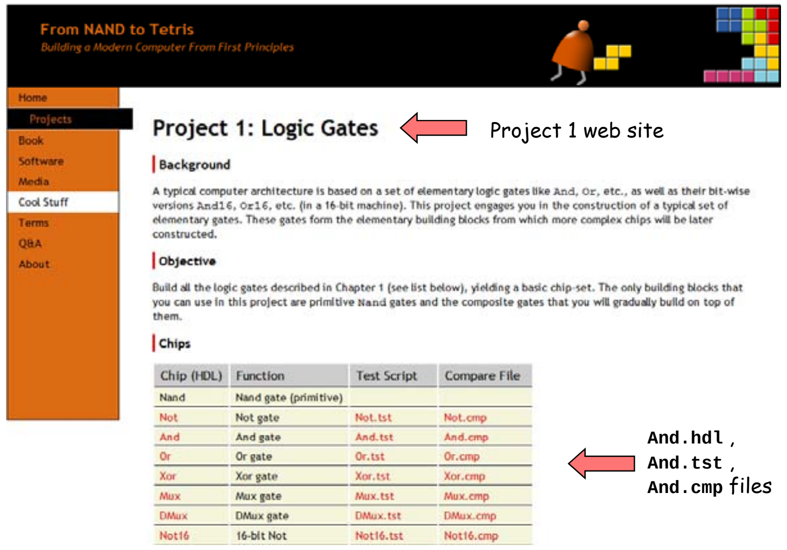
Name	Value
nota	0
notb	0
x	0
y	0

```
load Xor,
output-file Xor.out,
compare-to Xor.cmp,
output-list a%B3.1.3 b%B3.1.3 out%B3.1.3;

set a 0,
set b 0,
eval,
output;

set a 0,
set b 1,
eval,
output;

set a 1,
set b 0,
eval,
output;

set a 1,
set b 1,
eval,
output;
```

```
|   a   |   b   |  out  |
|   0   |   0   |   0   |
|   0   |   1   |   1   |
|   1   |   0   |   1   |
|   1   |   1   |   0   |
```

End of script - Comparison ended successfuly

Project materials: www.nand2tetris.org



Project 1: Logic Gates

Background

A typical computer architecture is based on a set of elementary logic gates like And, Or, etc., as well as their bit-wise versions And16, Or16, etc. (in a 16-bit machine). This project engages you in the construction of a typical set of elementary gates. These gates form the elementary building blocks from which more complex chips will be later constructed.

Objective

Build all the logic gates described in Chapter 1 (see list below), yielding a basic chip-set. The only building blocks that you can use in this project are primitive Nand gates and the composite gates that you will gradually build on top of them.

Chips

Chip (HDL)	Function	Test Script	Compare File
Nand	Nand gate (primitive)		
Not	Not gate	Not.tst	Not.cmp
And	And gate	And.tst	And.cmp
Or	Or gate	Or.tst	Or.cmp
Xor	Xor gate	Xor.tst	Xor.cmp
Mux	Mux gate	Mux.tst	Mux.cmp
DMux	DMux gate	DMux.tst	DMux.cmp
Not16	16-bit Not	Not16.tst	Not16.cmp

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.
```

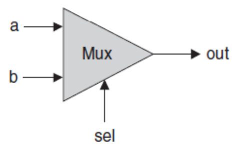
Gates for project #1

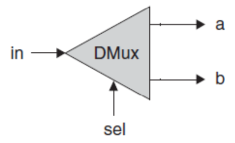
```
Chip name: Mux
Inputs:    a, b, sel
Outputs:   out
Function:  If sel=0 then out=a else out=b.
```

```
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

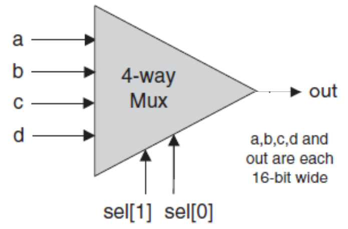


Figure 1.10 4-way multiplexor. The width of the input and output buses may vary.

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

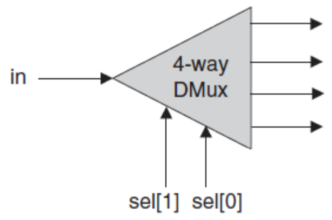


Figure 1.11 4-way demultiplexor.

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]".
```

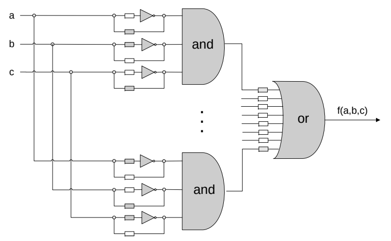
Gates for project #1 (Multi-way version)

```
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}.
```

Perspective

- Each Boolean function has a canonical representation
- The canonical representation is expressed in terms of And, Not, Or
- And, Not, Or can be expressed in terms of Nand alone
- Ergo, every Boolean function can be realized by a standard PLD consisting of Nand gates only
- Mass production
- Universal building blocks, unique topology
- Gates, neurons, atoms, …



End notes: Canonical representation

Whodunit story: Each suspect may or may not have an alibi (*a*), a motivation to commit the crime (*m*), and a relationship to the weapon found in the scene of the crime (*w*). The police decides to focus attention only on suspects for whom the proposition Not(*a*) And (*m* Or *w*) is true.

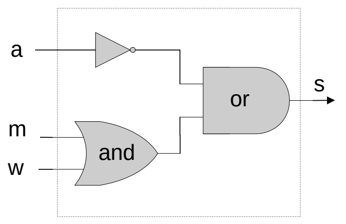
Truth table of the "suspect" function $s(a,m,w) = \bar{a} \cdot (m + w)$

a	m	w	*minterm*	suspect(a,m,w)= not(a) and (m or w)
0	0	0	$m_0 = \bar{a}\bar{m}\bar{w}$	0
0	0	1	$m_1 = \bar{a}\bar{m}w$	1
0	1	0	$m_2 = \bar{a}m\bar{w}$	1
0	1	1	$m_3 = \bar{a}mw$	1
1	0	0	$m_4 = a\bar{m}\bar{w}$	0
1	0	1	$m_5 = a\bar{m}w$	0
1	1	0	$m_6 = am\bar{w}$	0
1	1	1	$m_7 = a\,m\,w$	0

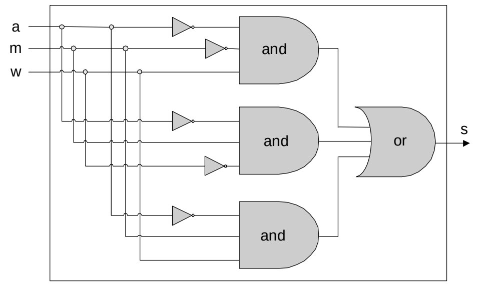
Canonical form: $s(a,m,w) = \bar{a}\,\bar{m}\,w + \bar{a}\,m\,\bar{w} + \bar{a}\,m\,w$

End notes: Canonical representation (cont.)

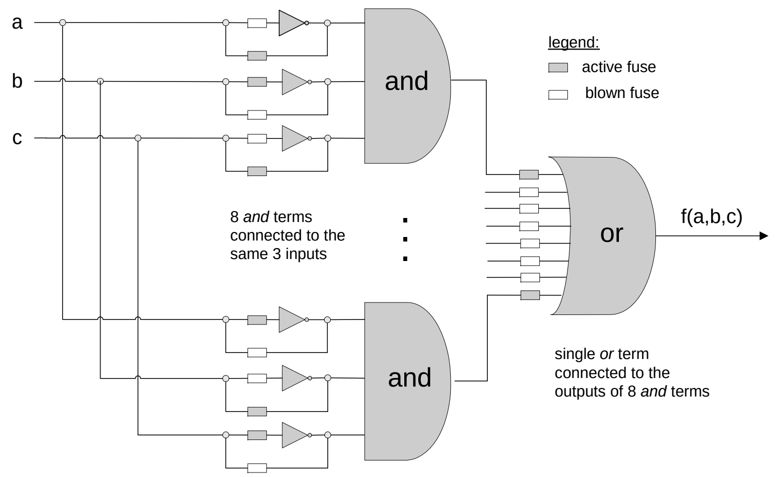
$$s(a,m,w) = \bar{a} \cdot (m + w)$$



$$s(a,m,w) = \bar{a}\,\bar{m}\,w + \bar{a}\,m\,\bar{w} + \bar{a}\,m\,w$$



End notes: Programmable Logic Device for 3-way functions

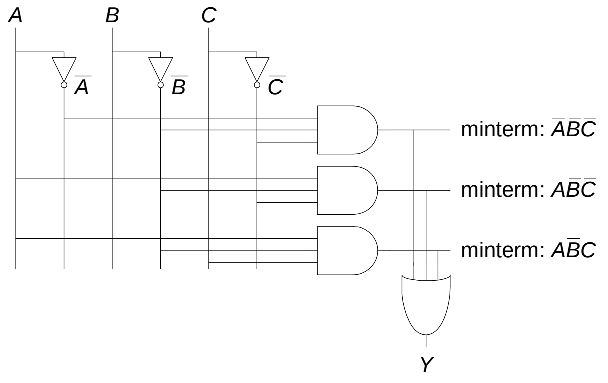

a
b
c
and
and
or
f(a,b,c)
legend:
▨ active fuse
▭ blown fuse
8 *and* terms connected to the same 3 inputs
single *or* term connected to the outputs of 8 *and* terms

PLD implementation of $f(a,b,c) = a\,\bar{b}\,c + \bar{a}\,b\,\bar{c}$

(the on/off states of the fuses determine which gates participate in the computation)

End notes: Programmable Logic Device for 3-way functions

- Two-level logic: ANDs followed by ORs
- Example: $Y = ABC + ABC + ABC$


A
B
C
$\bar{A}$
$\bar{B}$
$\bar{C}$
minterm: $\bar{A}\bar{B}\bar{C}$
minterm: $A\bar{B}\bar{C}$
minterm: $A\bar{B}C$
Y