

# Lecture 2 – Boolean Logic

Dr Tianxiang Cui

# Research interests - Tianxiang

- Operation Research
  - Deal with the development and application of advanced analytical methods to improve decision-making
- Computational Intelligence
  - Set of nature-inspired computational methodologies and approaches to address complex real-world problems
- Machine Learning
  - An area of AI concerned with development of techniques which allow machines to learn
- Reinforcement Learning
  - Framework for learning to solve sequential decision making problems



# Question

- “How do I get the text book”
- Library
- Book shop
- Individual chapter available online
- [https://docs.wixstatic.com/ugd/44046b\\_f2c9e41f0b204a34ab78be0ae4953128.pdf](https://docs.wixstatic.com/ugd/44046b_f2c9e41f0b204a34ab78be0ae4953128.pdf)
- [https://docs.wixstatic.com/ugd/44046b\\_89c60703ebfc4bf39acef13bdc050f5d.pdf](https://docs.wixstatic.com/ugd/44046b_89c60703ebfc4bf39acef13bdc050f5d.pdf)
- Whole digital version of the book may still be available

# Outline

- Boolean logic
- Boolean function synthesis
- Hardware description language (HDL)
- Hardware simulation
- Multi-bit buses

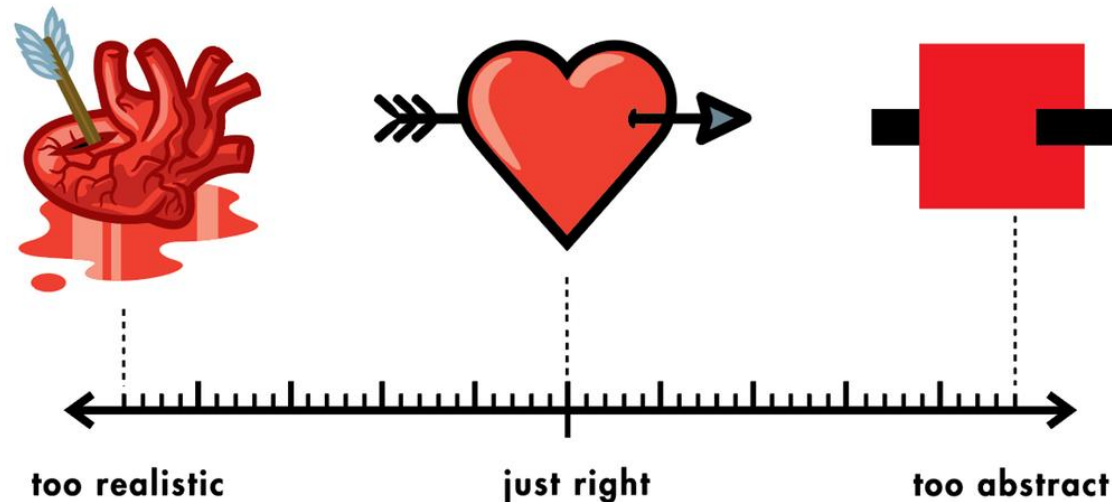
# Learning Outcome

- To be able to understand elementary logic gates
- To be able to simplify Boolean expression
- To be able to implement logic gates in HDL

# Abstractions

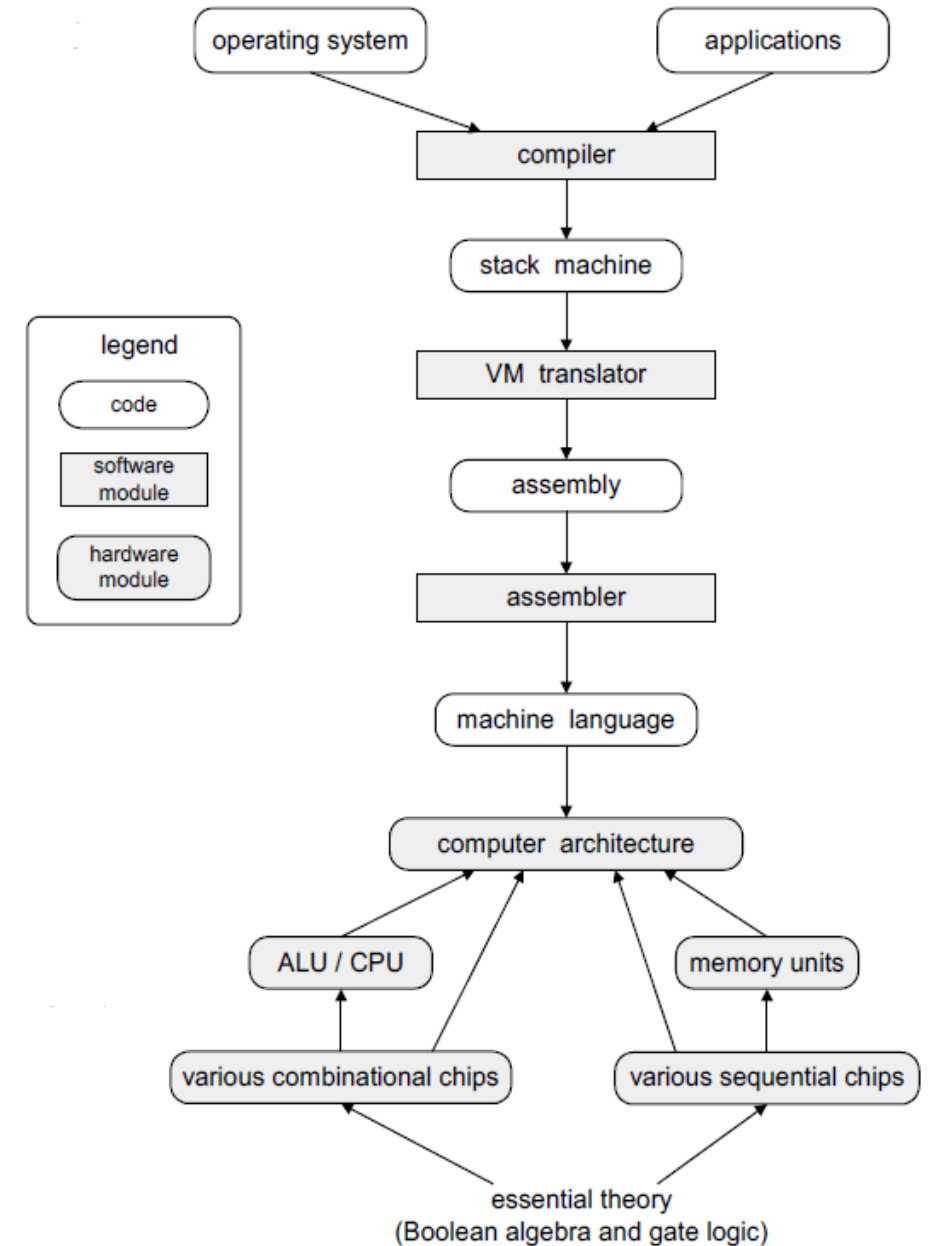
- Abstraction is a technique for arranging complexity of computer systems
  - “what the module does” not “how the module does it”
- Establish a level of complexity, and suppress the more complex details below the current level

## THE ABSTRACT-O-METER



# Top-down vs Bottom-up

- Top-down
  - Higher level abstractions can be expressed by simpler ones
- Bottom-up
  - Lower-Level abstractions used to construct more complex ones



# Top-down vs Bottom-up

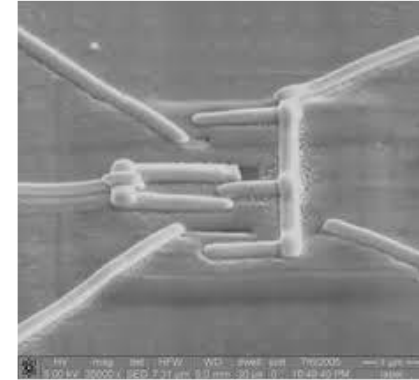
- TD vs BU occurs in many areas of computer science (and elsewhere)
  - Software development (Waterfall vs Agile)
  - Programming (Procedural vs OOP)
  - Simulation (Discrete event vs Agents)
- Important to know the difference and when to use what



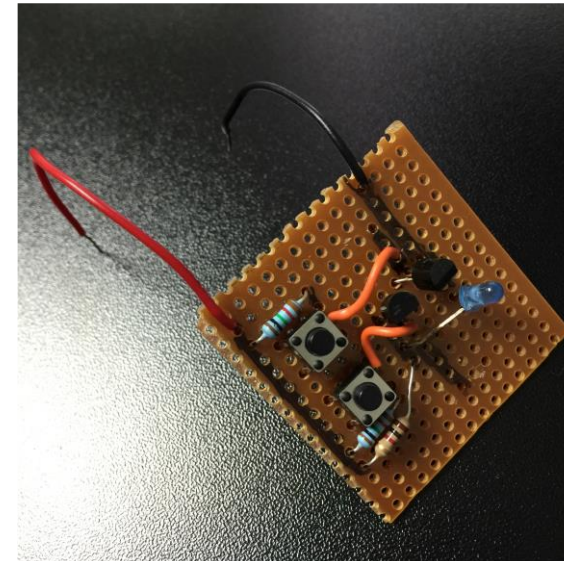


# Building from Elementary Components

- Components inside a computer is build up of billions of nano sized transistors
- The transistor is a basic building block used to construct more complex electronic components



Nano size transistor under microscope



Transistors on a Circuits Board With other electronic components connected as a NAND Gate

# Building from Elementary Components

- The CPU, RAM, ROM are built up from nano sized transistors



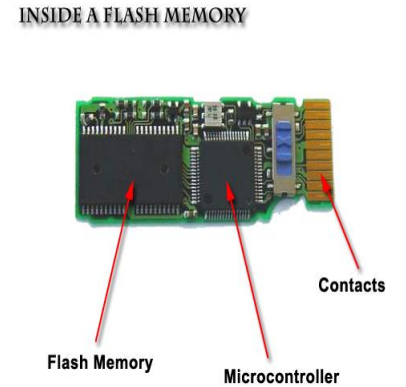
Central Processing Unit (CPU)



Erasable Programmable Read Only Memory (EPROM)



Random Access Memory (RAM)



Flash Memory

*Such simple things,  
And we make of them something so complex  
it defeats us,  
Almost.*

~ John Ashbery (b.1927), American Poet

# Boolean (Binary) Values



Off

False

N

0

Black



On

True

Y

1

White

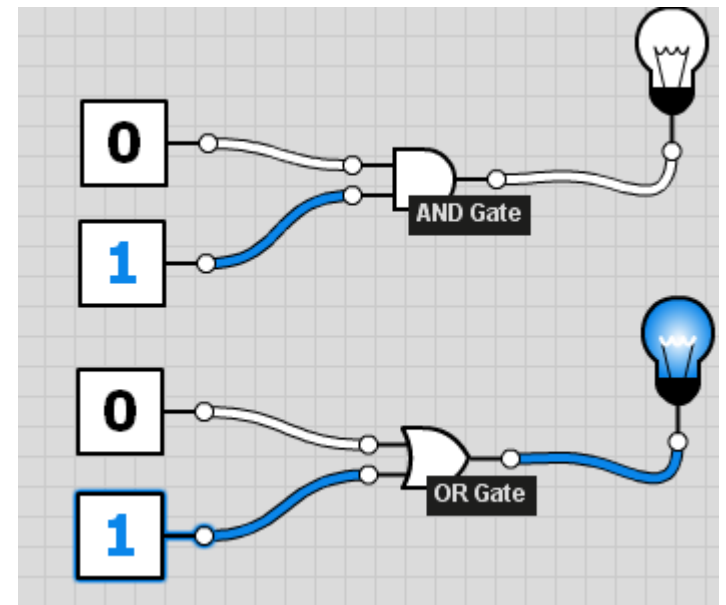


Colouring picture for ~~lazy~~ people.

**Boolean**

# Electronics

- We don't use physical logic gates, just simulations
  - We are not EEE!
- But always remember the intention
- Power will or will not flow through a circuit



# Boolean Logic

## All chips constructed from elementary logic gates

- Every chip can be built from a combination of:
  - AND
  - OR
  - NOT
  - No integration, division, differentiation...
  - “**Canonical Representation**”
- AND, OR and NOT can be built from NAND
  - We will see this later
- **Therefore every possible chip can be built from just the NAND gates**



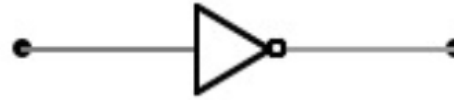
George Boole, 1815-1864  
(*"A Calculus of Logic"*)

# Boolean Function

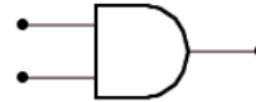
- A Boolean function is a function that operates on binary inputs and return binary outputs
- Truth table is **every possible function evaluation** of the input variables
- [note 0 and 1 used to define false and true]
- Everything can be defined by a truth table

# Elementary Logic Gates

$$A = \bar{A}$$



$$A \text{ AND } B = A \cdot B$$



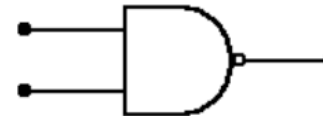
$$A \text{ OR } B = A + B$$



$$A \text{ XOR } B = A \oplus B$$



$$A \text{ NAND } B = \overline{A \cdot B}$$



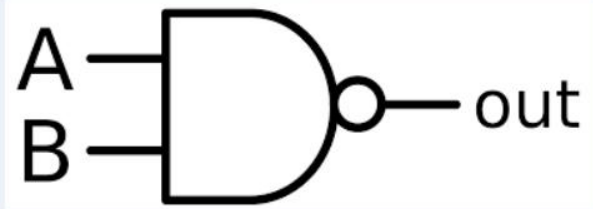
$$A \text{ NOR } B = \overline{A + B}$$





# Representations of Simple Logic Gate

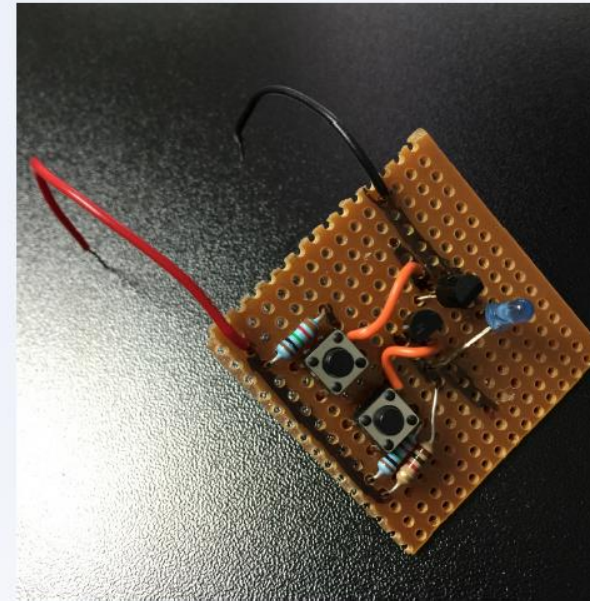
They are all the same!



Logic Gate Diagram

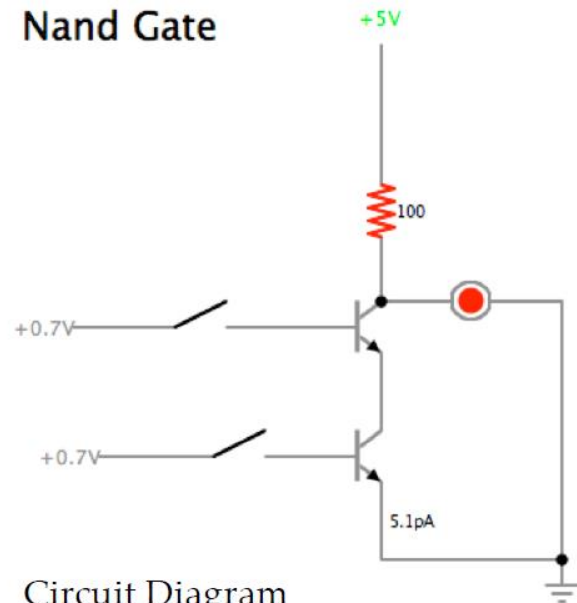
A1	A2	X1
0	0	1
0	1	1
1	0	1
1	1	0

Truth Table



Transistors on a Circuits Board With other electronic components connected as a NAND Gate.

Nand Gate



Circuit Diagram

$$f(A, B) = \overline{A \cdot B}$$

# NOT gate

- NOT gate – inverter – if 0 then 1
- (important, only use **one** entry/power supply)
- The “bubble” (o) at the end of the NOT gate symbol denotes a signal inversion (complementation) of the output signal.



A	$\bar{A}$
0	1

$$f(A) = \bar{A}$$

# AND gate

- If A and B is true then (A.B) is true, otherwise false

$$A \text{ AND } B = A \cdot B$$



A	B	$A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

$$f(A, B) = A \cdot B$$

# OR gate

- If A or B is true then (A+B) is true, otherwise false

$$A \text{ OR } B = A + B$$



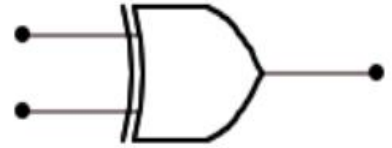
A	B	$A + B$
0	0	0
0	1	1
1	0	1
1	1	1

$$f(A, B) = A + B$$

# XOR gate

- Exclusive or
- Is true if and only if the two inputs are the different, otherwise false

$$A \text{ XOR } B = A \oplus B$$



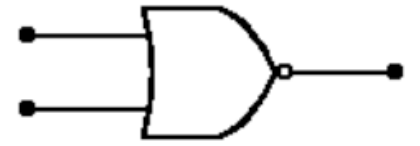
A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

$$f(A, B) = A \oplus B$$

# NOR gate

- Negation of OR gate
- Is true if and only if the two inputs are false, otherwise false

$$A \text{ NOR } B = \overline{A + B}$$



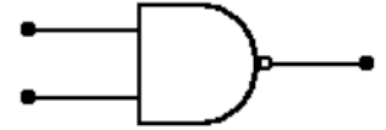
A	B	$\overline{A + B}$
0	0	1
0	1	0
1	0	0
1	1	0

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

# NAND gate

- Negation of AND gate
- Is false if and only if the two inputs are true, otherwise true

$$A \text{ NAND } B = \overline{A \cdot B}$$



A	B	$\overline{A \cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0

$$f(A, B) = \overline{A \cdot B}$$

# Collection of Elementary Logic Gates

$$A = \bar{A} \quad \text{---} \triangle \text{---}$$

A	$\bar{A}$
0	1

$$A \text{ XOR } B = A \oplus B \quad \text{---} \text{---} \text{---} \text{---} \text{---}$$

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

$$A \text{ AND } B = A \cdot B \quad \text{---} \text{---} \text{---} \text{---}$$

A	B	$A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

$$A \text{ NAND } B = \overline{A \cdot B} \quad \text{---} \text{---} \text{---} \text{---} \text{---}$$

A	B	$\overline{A \cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0

$$A \text{ OR } B = A + B \quad \text{---} \text{---} \text{---} \text{---}$$

A	B	$A + B$
0	0	0
0	1	1
1	0	1
1	1	1

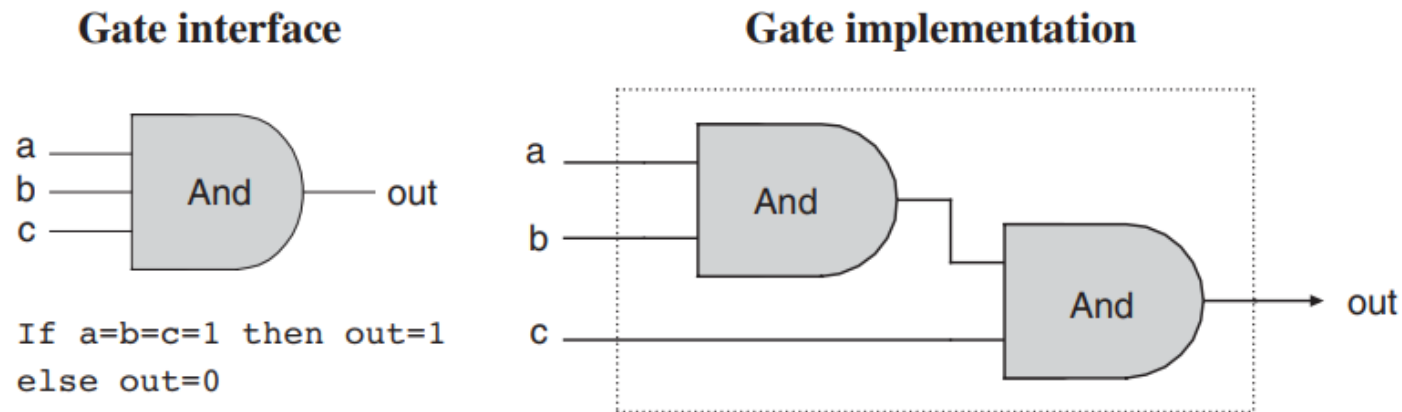
$$A \text{ NOR } B = \overline{A + B} \quad \text{---} \text{---} \text{---} \text{---} \text{---}$$

A	B	$\overline{A + B}$
0	0	1
0	1	0
1	0	0
1	1	0



# Gate Logic

- Gate is a physical device to implement Boolean logic
- Elementary gates only have 1 or 2 inputs
- Gates with 3 or more inputs require composing a structure with multiple gates (hence called composite gates)



**Figure 1.4** Composite implementation of a three-way And gate. The rectangle on the right defines the conceptual boundaries of the gate interface.

# Composite Gates

- What gate is the truth table on the right?

$$f(A, B) = \overline{A \cdot B}$$

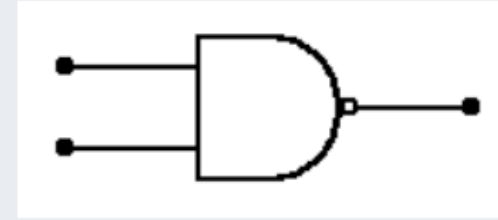


A	B	$f(A, B)$
0	0	1
0	1	1
1	0	1
1	1	0

# Composite Gates

- What gate is the truth table on the right?

**It's the NAND Gate!**

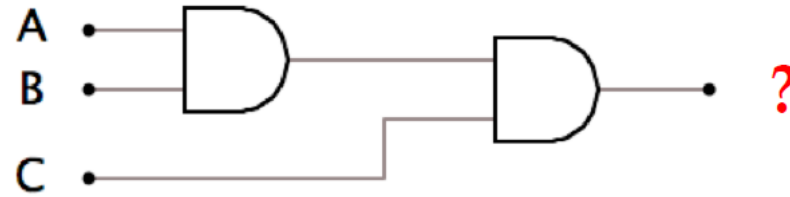


A	B	$\overline{A \bullet B}$
0	0	1
0	1	1
1	0	1
1	1	0

# Composite Gates

Three Input And Gates

$$f(A,B,C) = A \bullet B \bullet C$$

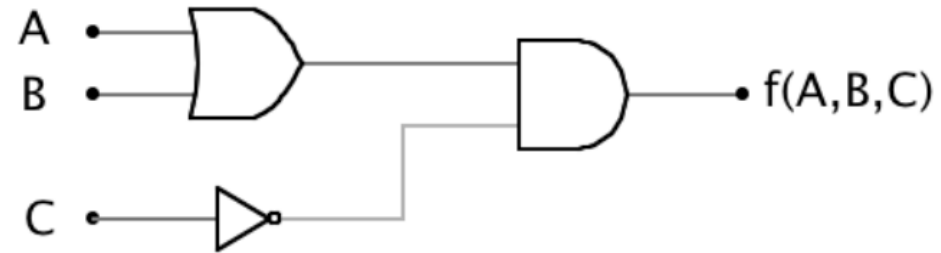


A	B	C	$f(A,B,C)$
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

# Composite Gates

$$f(A, B, C) = (A + B) \cdot \overline{C}$$

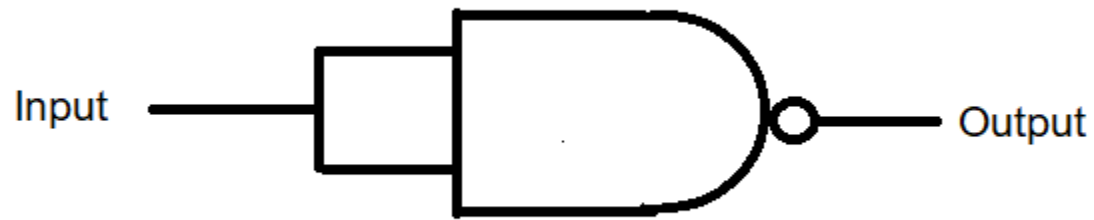
(A OR B) AND NOT C



A	B	C	$f(A, B, C)$
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

# Composite Gates

- What truth table is this?



A	B	$f(A,B)$
0	0	1
1	1	0

- NOT gate can be made using a NAND gate by connecting both inputs of the gate to the single input signal

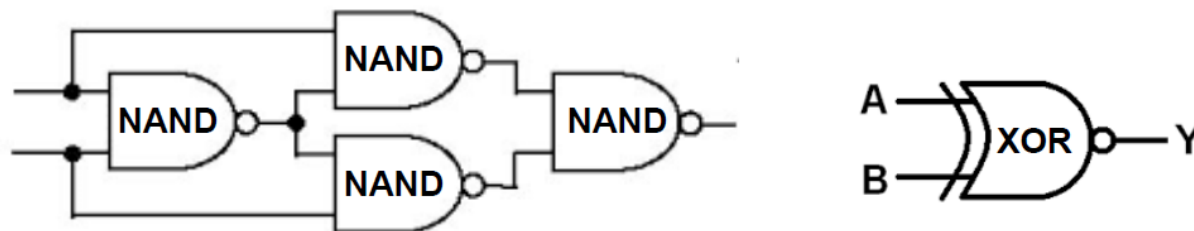
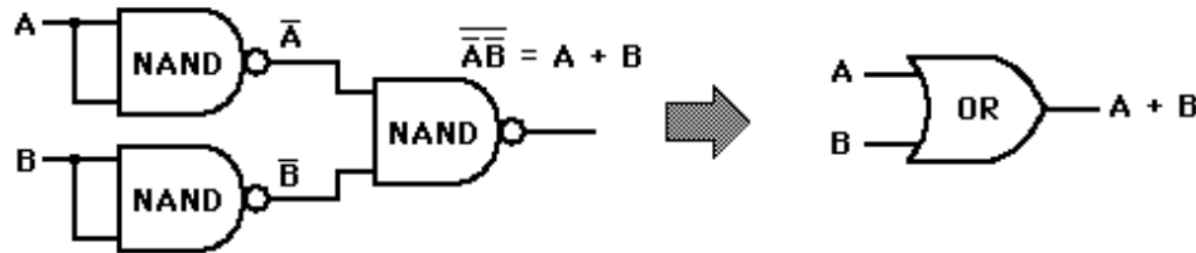
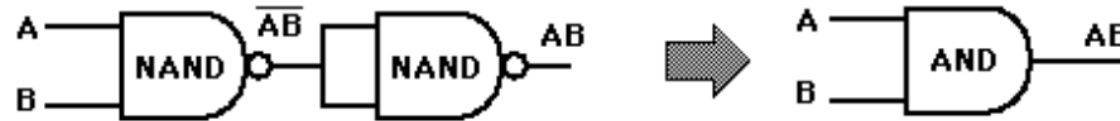
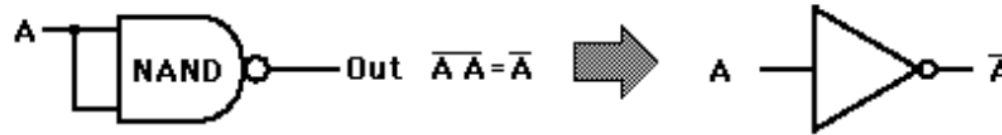
# The Super NAND

- All Boolean logic can be built from And, Or and Not
  - Because it is possible to implement all of the possible Boolean switching functions
  - 'Truth table to Boolean expression' method just uses these three operations
  - Truth table can define anything
- All Boolean logic can be built from And and Not
  - Because...
  - $x \text{ Or } y = \text{Not}(\text{Not}(x) \text{ And } \text{Not}(y))$
- All Boolean logic can be built using Nand gates
  - Because....
  - $\text{Not}(x) = \text{Nand}(x, x)$
  - $\text{And}(x, y) = \text{Not}(\text{Nand}(x, y))$
  - $\text{Or}(x, y) = (\text{Nand}(\text{Nand}(x, y), \text{Nand}(x, y)))$

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

# The Super NAND

- The NAND gate is universal and can be used to construct all other gates





# Boolean Expressions

Produce a Boolean value when evaluated

**$X = 0, Y = 1, Z = 1$**

Not(X Or (Y And Z))

Not(0 Or (1 And 1)) =

Not(0 Or 1) =

Not(1) = 0

Not(X) Or (Y And Z)

Not(0) Or (1 And 1) =

Not(0) Or (1) =

1 Or 1 = 1

(Brackets  
Matter)

# Precedence

- Precedence

**Parentheses** evaluated first

Then **Not**

Then **And**

Then **Or**

**Not X Or Y And Z = (Not X) Or (Y And Z)**

**Not X And Y Or Z = ((Not X) And Y) Or Z**

Brackets over-rule everything...use when in doubt

**((Not (X)) And (Y)) Or (Z)**

# Boolean Expression – using precedence

$$f(x,y,z) = \text{Not}(a) \text{ Or}(b) \text{ And}(c)$$

What is  $f(x,y,z)$  when  $a=1$ ,  $b=0$ ,  $c=1$ ?

$$= (\text{Not}(a)) \text{ Or } (b) \text{ And } (c)$$

$$= 0 \text{ Or } ((b) \text{ And } (c))$$

$$= 0 \text{ Or } 0$$

$$f(x,y,z) = 0$$

# Boolean Functions

$$f(x, y, z) = (x \text{ And } y) \text{ Or } (\text{Not}(x) \text{ And } z)$$

# Boolean Functions

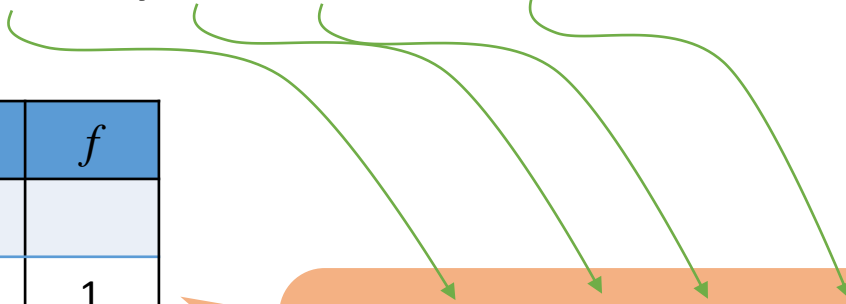
$$f(x, y, z) = (x \text{ And } y) \text{ Or } (\text{Not}(x) \text{ And } z)$$

$x$	$y$	$z$	$f$
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

# Boolean Functions

$$f(x, y, z) = (x \text{ And } y) \text{ Or } (\text{Not}(x) \text{ And } z)$$


$x$	$y$	$z$	$f$
0	0	0	
0	0	1	1
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	1
1	1	1	



$(0 \text{ And } 0) \text{ Or } (\text{Not}(0) \text{ And } 1) =$   
 $0 \text{ Or } (1 \text{ And } 1) =$   
 $0 \text{ Or } 1 = 1$

$(1 \text{ And } 1) \text{ Or } (\text{Not}(1) \text{ And } 0) =$   
 $1 \text{ Or } (0 \text{ And } 0) =$   
 $1 \text{ Or } 0 = \dots$

# Boolean Functions

$$f(x, y, z) = (x \text{ And } y) \text{ Or } (\text{Not}(x) \text{ And } z)$$


formula

$x$	$y$	$z$	$f$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



truth table

# Laws of Boolean Algebra

- **Idempotent law:**  $x \text{ Or } x = x$      $x \text{ And } x = x$      $x \text{ Or } x \text{ Or } x \text{ Or } x \dots = x$ 
  - Operation can be applied multiple times without changing the result beyond the initial application
- **Associative law:**  $(x \text{ Or } y) \text{ Or } z = x \text{ Or } (y \text{ Or } z)$      $(x \text{ And } y)z = x(y \text{ And } z)$ 
  - Terms may be associated in any way desired if **same logical operations** used
  - Because precedence is the same!
  - (obviously doesn't work if you mix Or/And)

And is implicit





# Laws of Boolean Algebra

- **Commutative law:**  $x \text{ And } y = y \text{ And } x$   $x \text{ Or } y = y \text{ Or } x$ 
  - Function outcome is unaltered by reordering its terms

- **Distributive law:**  $x \text{ And } (y \text{ Or } z) = (x \text{ And } y) \text{ Or } (x \text{ And } z)$  (OR Distributive Law)

- $x \text{ Or } (y \text{ And } z) = (x \text{ Or } y) \text{ And } (x \text{ Or } z)$  (AND Distributive Law)

same as normal math when you expand out an equation

# Laws of Boolean Algebra

- **De Morgans Law:**

- **$\text{Not}(x \text{ And } y) = \text{Not}(x) \text{ Or } \text{Not}(y)$  [1]**
- **$\text{Not}(x \text{ Or } y) = \text{Not}(x) \text{ And } \text{Not}(y)$  [2]**

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

$[1]-l$	$[l]-r$
1	1
1	1
1	1
0	0

$[2]-l$	$[2]-r$
1	1
0	0
0	0
0	0

# Laws of Boolean Algebra

- **Complement law:  $x \text{ And}(\text{Not}(x))=0$     $x \text{ Or} (\text{Not}(x))=1$** 
  - A term **And** with its complement equals 0 and a term **Or** with its complement equals 1
- **Double negative Law:  $(\text{NOT}(\text{NOT}(x)))=x$** 
  - A function, when applied twice, brings one back to the starting point. (double negation)

# Laws of Boolean Algebra

## 1. Law of Identity

$$A = A$$

$$\bar{A} = \bar{A}$$

## 2. Commutative Law

$$A \cdot B = B \cdot A$$

$$A + B = B + A$$

## 3. Associative Law

$$(A \cdot B) \cdot C = A \cdot (B \cdot C)$$

$$(A + B) + C = A + B + C$$

## 4. Idempotent Law

$$A \cdot A = A$$

$$A + A = A$$

## 5. Double Negative Law

$$\bar{\bar{A}} = A$$

## 6. Complementary Law

$$A \cdot \bar{A} = 0$$

$$A + \bar{A} = 1$$

## 7. Law of Intersection

$$A \cdot 1 = A$$

$$A \cdot 0 = 0$$

## 8. Law of Union

$$A + 1 = 1$$

$$A + 0 = A$$

## 9. Distributive Law

$$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$$

$$A + (B \cdot C) = (A + B) \cdot (A + C)$$

## 10. Law of Absorption

$$A \cdot (A + B) = A$$

$$A + A \cdot B = A$$

## 11. Law of Common Identities

$$A \cdot (\bar{A} + B) = AB$$

$$A + (\bar{A} \cdot B) = A + B$$

## 12. De Morgan's Law

$$\overline{A \cdot B} = \bar{A} + \bar{B}$$

$$\overline{A + B} = \bar{A} \cdot \bar{B}$$

# Boolean Algebra

Not(Not(x) And Not(x Or y)) =

# Boolean Algebra

De Morgan law

Not(Not(x) And **Not(x Or y)**) =

Not(Not(x) And **(Not(x) And Not(y))**) =

# Boolean Algebra

$\text{Not}(\text{Not}(x) \text{ And } \text{Not}(x \text{ Or } y)) =$

$\text{Not}(\text{Not}(x) \text{ And } (\text{Not}(x) \text{ And } \text{Not}(y))) =$

$\text{Not}((\text{Not}(x) \text{ And } \text{Not}(x)) \text{ And } \text{Not}(y)) =$

associative law (it doesn't matter what order we do our Ands in)

# Boolean Algebra

$\text{Not}(\text{Not}(x) \text{ And } \text{Not}(x \text{ Or } y)) =$

$\text{Not}(\text{Not}(x) \text{ And } (\text{Not}(x) \text{ And } \text{Not}(y))) =$

$\text{Not}(\text{Not}(x) \text{ And } \text{Not}(x)) \text{ And } \text{Not}(y) =$

$\text{Not}(\text{Not}(x)) \text{ And } \text{Not}(y) =$

Idempotence – doesn't matter how many times we And the Not(x)



# Boolean Algebra

$\text{Not}(\text{Not}(x) \text{ And } \text{Not}(x \text{ Or } y)) =$

$\text{Not}(\text{Not}(x) \text{ And } (\text{Not}(x) \text{ And } \text{Not}(y))) =$

$\text{Not}((\text{Not}(x) \text{ And } \text{Not}(x)) \text{ And } \text{Not}(y)) =$

$\text{Not}(\text{Not}(x) \text{ And } \text{Not}(y)) =$

De Morgan law (can use both ways around)

$\text{Not}(\text{Not}(x \text{ Or } y)) =$

# Simplify Boolean Expression

$\text{Not}(\text{Not}(x) \text{ And } \text{Not}(x \text{ Or } y)) =$

$\text{Not}(\text{Not}(x) \text{ And } (\text{Not}(x) \text{ And } \text{Not}(y))) =$

$\text{Not}((\text{Not}(x) \text{ And } \text{Not}(x)) \text{ And } \text{Not}(y)) =$

$\text{Not}(\text{Not}(x) \text{ And } \text{Not}(y)) =$

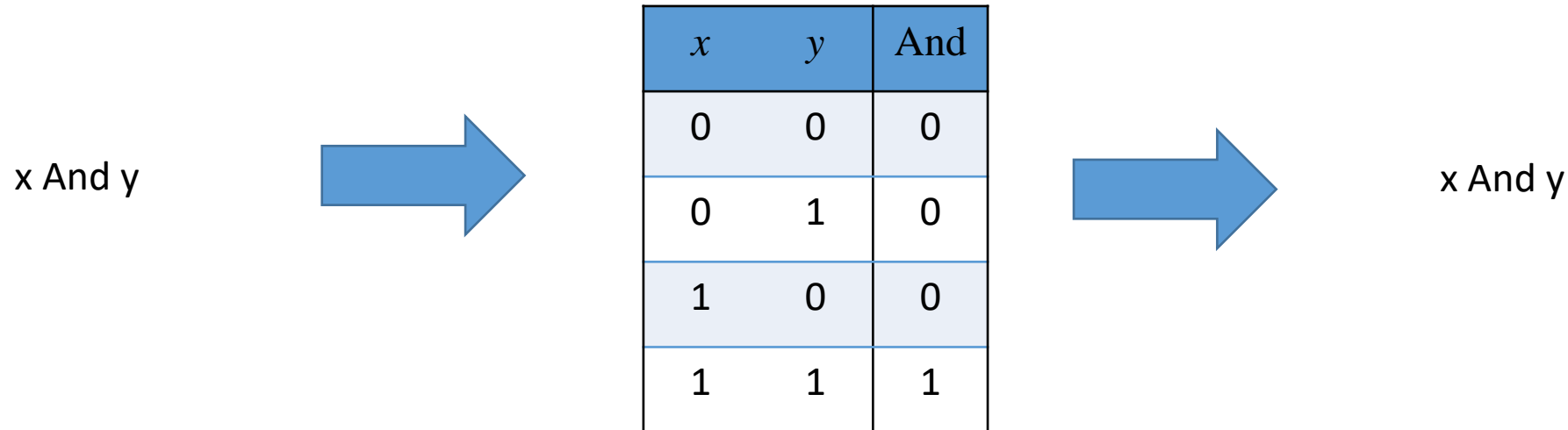
$\text{Not}(\text{Not}(x \text{ Or } y)) =$

double negation

$x \text{ Or } y$

# Boolean function synthesis

- We know how to convert a Boolean expression into a truth table..
- ..but how to convert truth table to a Boolean expression?



---

$x$	$y$	$z$	$f$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

From truth table to a Boolean expression

---

$x$	$y$	$z$	$f$
0	0	0	1 1
0	0	1	0 0
0	1	0	1 0
0	1	1	0 0
1	0	0	1 0
1	0	1	0 0
1	1	0	0 0
1	1	1	0 0

(Not(x) And Not(y) And Not(z))

---

$x$	$y$	$z$	$f$
0	0	0	1 0
0	0	1	0 0
0	1	0	1 1
0	1	1	0 0
1	0	0	1 0
1	0	1	0 0
1	1	0	0 0
1	1	1	0 0

(Not(x) And y And Not(z))

---

$x$	$y$	$z$	$f$
0	0	0	1 0
0	0	1	0 0
0	1	0	1 0
0	1	1	0 0
1	0	0	1 1
1	0	1	0 0
1	1	0	0 0
1	1	1	0 0

(x And Not(y) And Not(z))

---

$x$	$y$	$z$	$f$
0	0	0	1 1
0	0	1	0
0	1	0	1 1
0	1	1	0
1	0	0	1 1
1	0	1	0
1	1	0	0
1	1	1	0

(Not(x) And Not(y) And Not(z))

(Not(x) And y And Not(z))

(x And Not(y) And Not(z))



---

$x$	$y$	$z$	$f$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

(Not( $x$ ) And Not( $y$ ) And Not( $z$ ))

Or

(Not( $x$ ) And  $y$  And Not( $z$ ))

Or

( $x$  And Not( $y$ ) And Not( $z$ ))

(Not( $x$ ) And Not( $y$ ) And Not( $z$ )) Or  
(Not( $x$ ) And  $y$  And Not( $z$ )) Or  
( $x$  And Not( $y$ ) And Not( $z$ )) =

---

$x$	$y$	$z$	$f$
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

(Not( $x$ ) And Not( $y$ ) And Not( $z$ ))

Or

(Not( $x$ ) And  $y$  And Not( $z$ ))

Or

( $x$  And Not( $y$ ) And Not( $z$ ))

(Not( $x$ ) And Not( $y$ ) And Not( $z$ )) Or  
(Not( $x$ ) And  $y$  And Not( $z$ )) Or  
( $x$  And Not( $y$ ) And Not( $z$ )) =

**(Not( $x$ ) Or ( $x$  And Not( $y$ ))) And Not( $z$ )**

# We can simplify, but why?

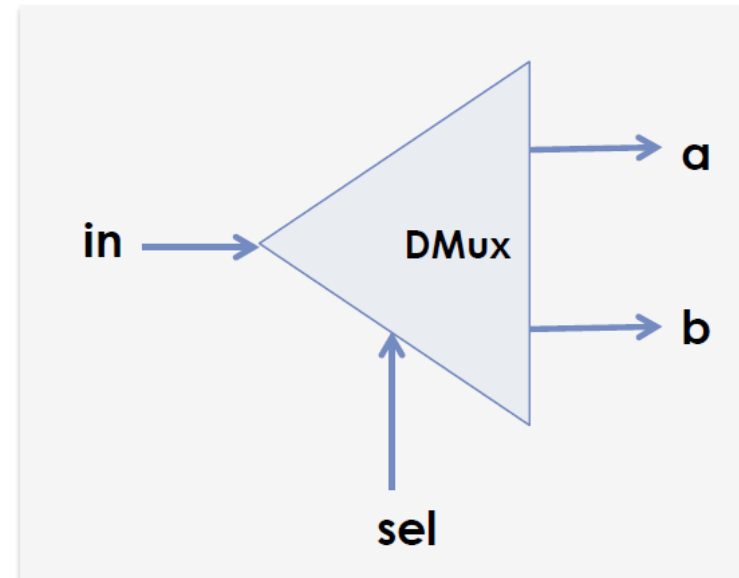
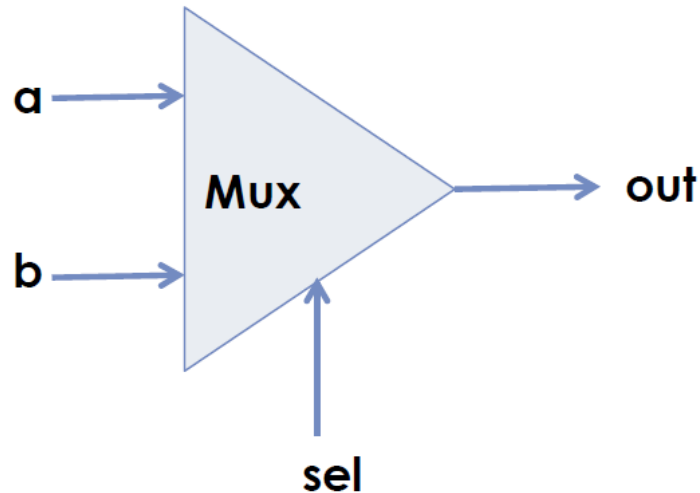
“everything should be as simple as it can be but not simpler”

Einstein

- $(\text{Not}(x) \text{ And } \text{Not}(y) \text{ And } \text{Not}(z)) \text{ Or } (\text{Not}(x) \text{ And } y \text{ And } \text{Not}(z)) \text{ Or } (x \text{ And } \text{Not}(y) \text{ And } \text{Not}(z))$
  - $(\text{Not}(x) \text{ Or } (x \text{ And } \text{Not}(y))) \text{ And } \text{Not}(z)$
  - Both correct
1. Simplicity / Transparency
  2. Less chips in silicon (cheaper, faster, less energy, less cycles, more robust)
  3. Impact on assignments and exams...

# Multiplexers and Demultiplexers

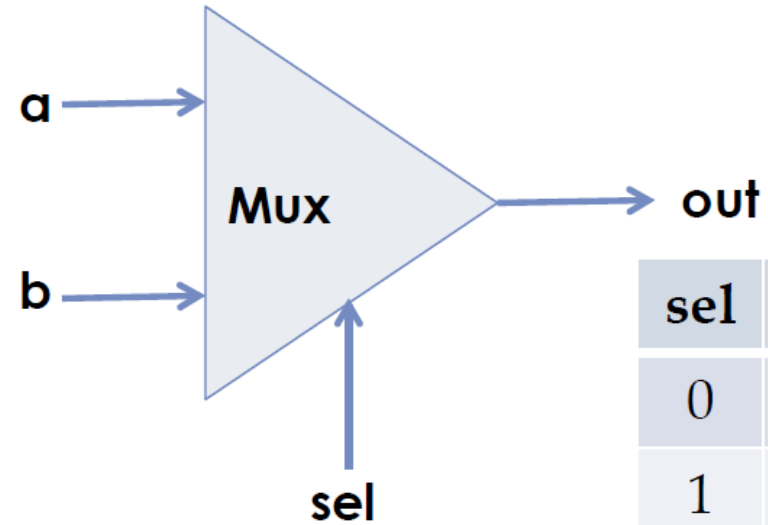
- A multiplexer is a combinational circuit that provides **single output** but accepts **multiple data inputs**.
- A demultiplexer is a combinational circuit that takes **single input** but that input can be directed through **multiple outputs**.



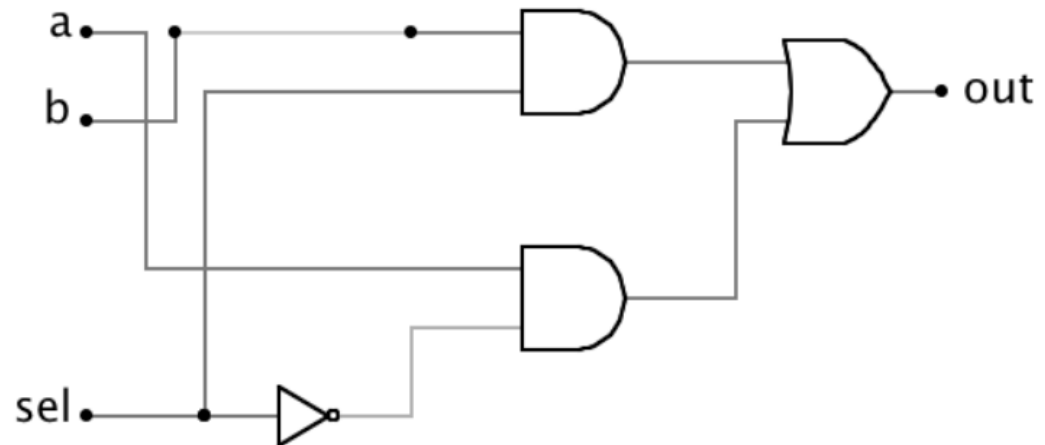
# Multiplexer

a	b	SEL	OUT
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

If sel==1 then  
out = b  
else  
out = a

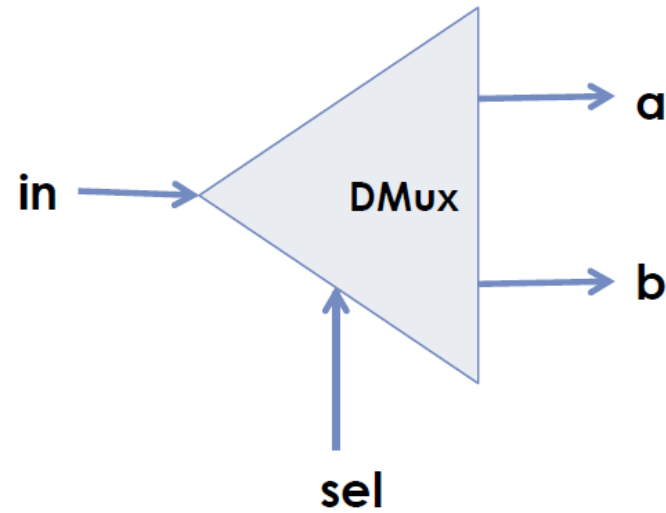


sel	out
0	a
1	b

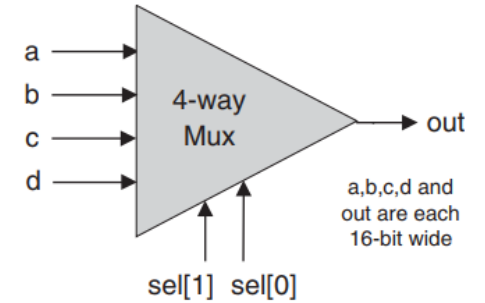


# Demultiplexer

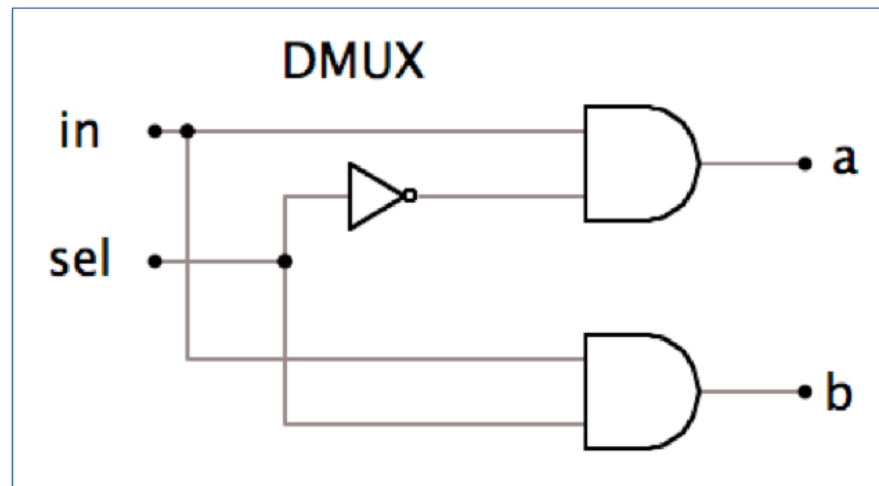
sel	a	b
0	in	0
1	0	in



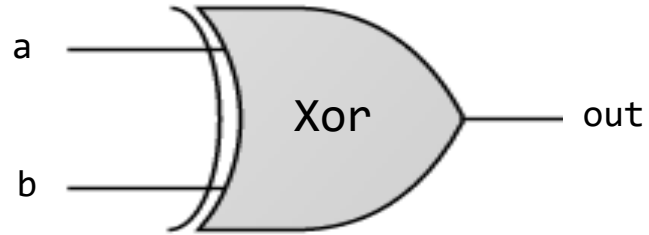
sel[1]	sel[0]	out
0	0	a
0	1	b
1	0	c
1	1	d



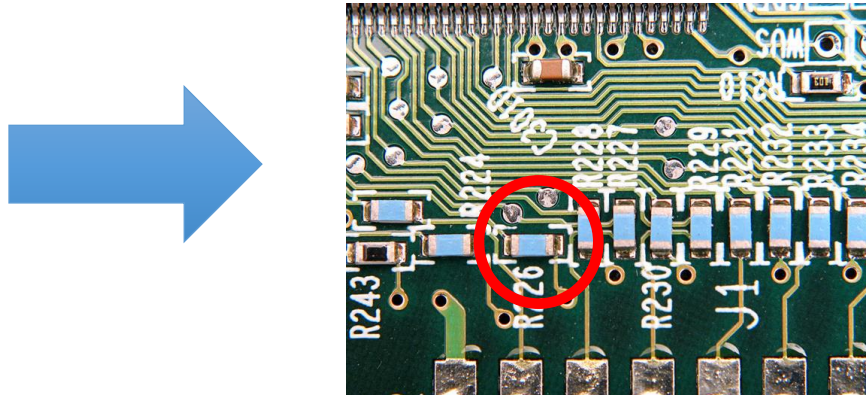
If  $\text{sel} = 0$  then  
     $\{a = \text{in}; b = 0\}$   
else  
     $\{a = 0; b = \text{in}\}$



# Building a logic gate



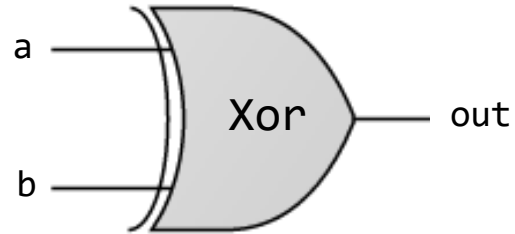
outputs 1 if one, and only one, of its inputs, is 1.



## The Process:

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

# Requirements to interface



Outputs 1 if one, and only one, of its inputs, is 1.

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

## Requirement:

Build a gate that delivers this functionality

```
/** Xor gate: out = (a And Not(b)) Or (Not(a) And b)) */
```

```
CHIP Xor {  
  IN a, b;  
  OUT out;
```

```
  PARTS:
```

```
    // Implementation missing
```

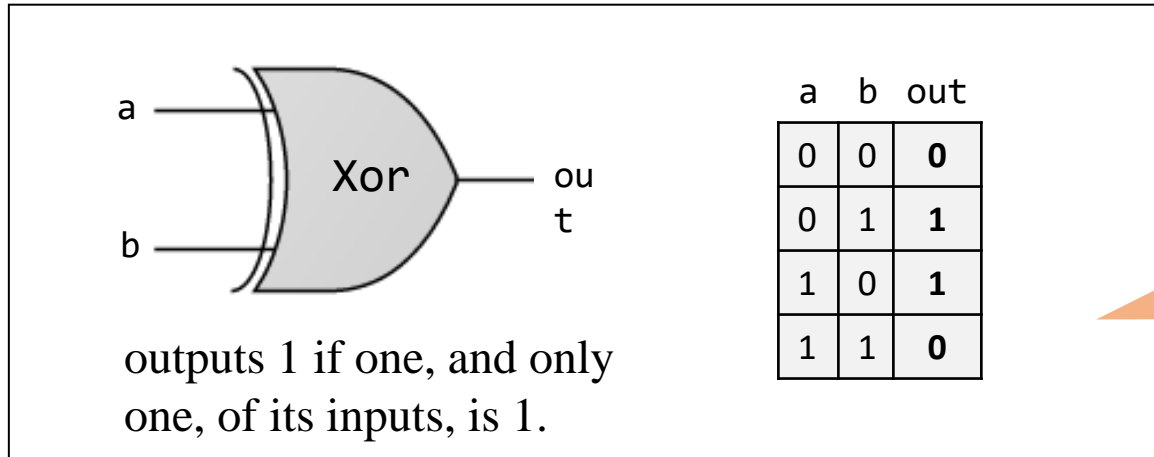
```
}
```

## Gate interface

Expressed as an HDL stub file



# Requirements to gate diagram



Requirement:

Build a gate that delivers this functionality



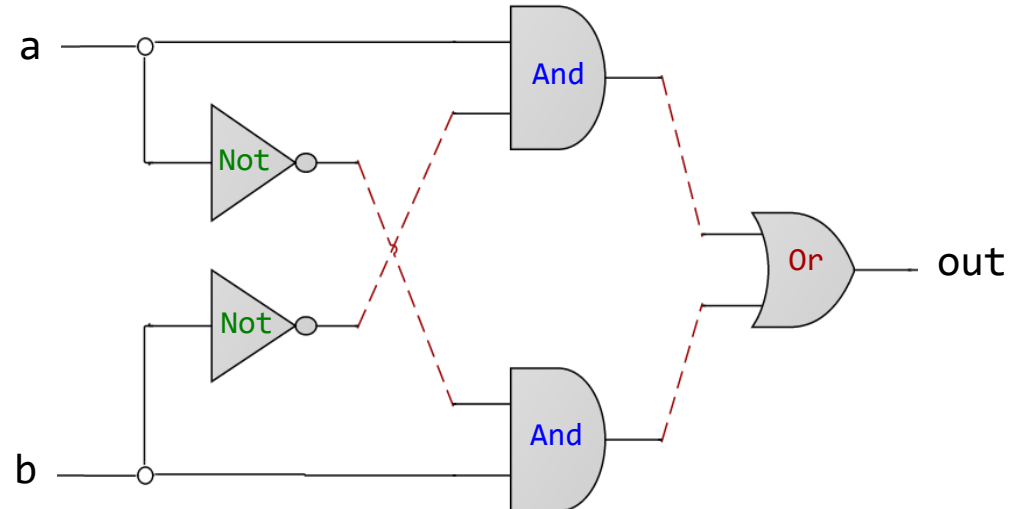
General idea:

out=1 when:

a And Not(b)

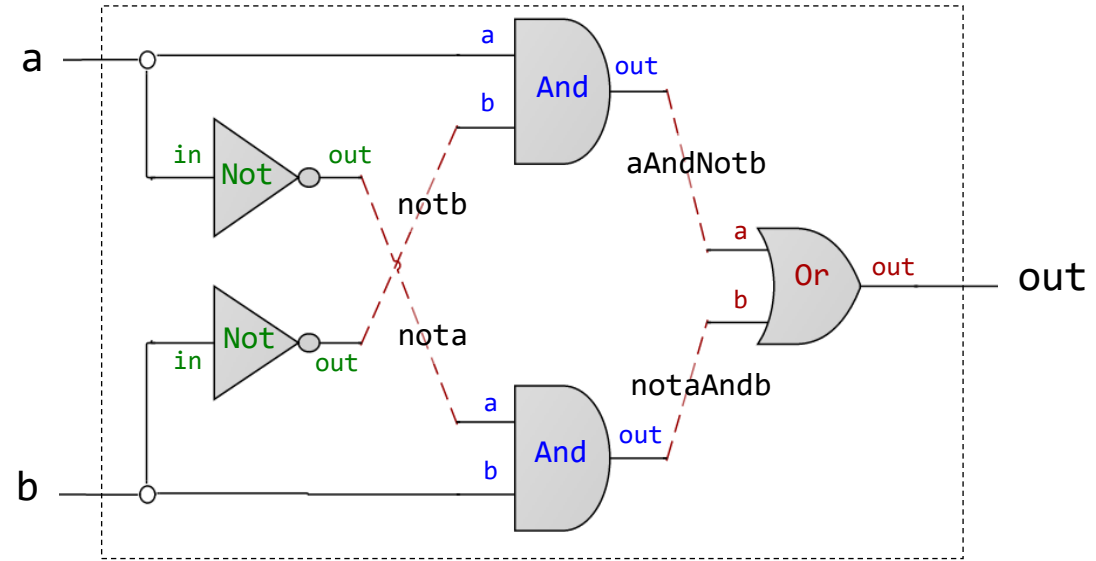
Or

b And Not(a)



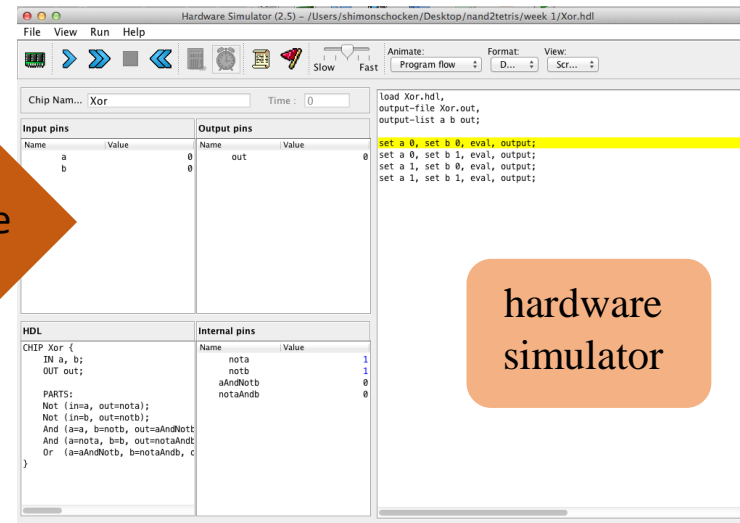
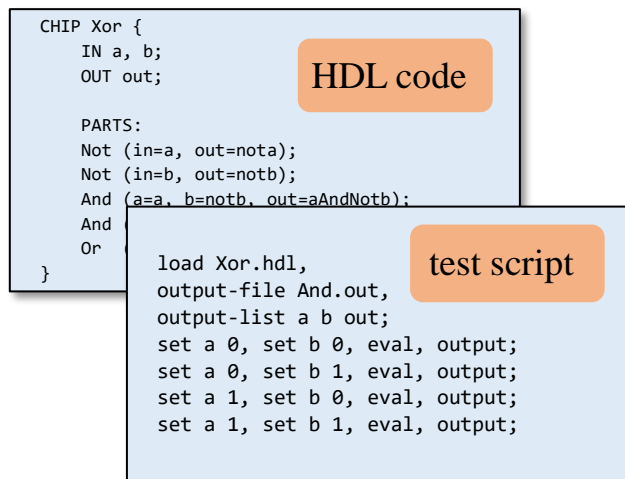
!Precedence

# Gate Diagram to HDL Code



```
interface {  
    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 Simulator



## 2 Options

- Load Chip – loads HDL code
- Load Script – loads testing script

## Simulation options:

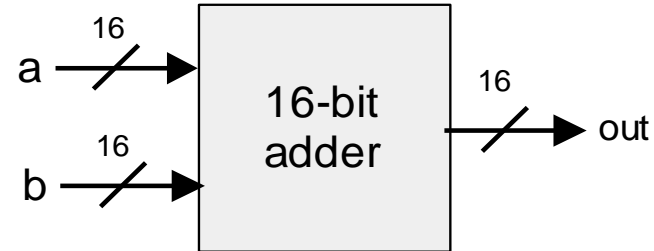
- Interactive
- Script-based
- With / without output and compare files

# Multi-Bit Buses

- Arrays of Bits
- Sometimes we wish to manipulate an array of bits as one group
- It's convenient to think about such a group of bits as **a single entity**, sometime termed "**bus**"
- HDL usually provide notation and means for handling buses.

# Example – Adding two or three 16 bit integers

```
/*  
 * Adds two 16-bit values.  
 */  
CHIP Add16 {  
    IN a[16], b[16];  
    OUT out[16];  
  
    PARTS:  
    ...  
}
```



```
/*  
 * Adds three 16-bit inputs.  
 */  
CHIP Add3Way16 {  
    IN first[16], second[16], third[16];  
    OUT out[16];  
  
    PARTS:  
    Add16(a=first, b=second, out=temp);  
    Add16(a=temp, b=third, out=out);  
}
```

# Working with single bits within an array

a[4] = 0100

(a[0]: right-most bit, a[3]: left-most bit)

```
/*
 * 4-way And: Ands 4 bits.
 */
CHIP And4Way {
    IN a[4]; / beware, element no a[4]
    OUT out;

    PARTS:
        And(a=a[0], b=a[1], out=t01);
        And(a=t01, b=a[2], out=t012);
        And(a=t012, b=a[3], out=out);
}
```

out = 0

a[4] = 0100 b[4] = 1101

```
/*
 * 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]);
}
```

out = 0100

# Summary

- Boolean logic
  - basics, truth tables, laws, function compression
- Boolean function synthesis
- Hardware description language (HDL)
  - gate design, code generation
- Hardware simulation
  - usage, test scripts, logic gates
- Multi-bit buses
  - arrays

# First Lab Session

- (Last year it took between 5 minutes and 2 weeks for people to get Nand2Tetris software to work)
- Download software packages from:
  - <https://www.nand2tetris.org/software>
  - <https://www.nand2tetris.org/course>
- Check you can run the HardwareSimulator that is in \nand2tetris\tools
- Check you can load a chip (eg. The xor chip)
  - \nand2tetris\projects\01\Xor.hdl
- Check you can edit the Xor.hdl file
- Labsheet will be supplied