Lecture 2 – Boolean Logic

Dr Tianxiang Cui

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_89c60703ebfc4bf39acef13bdc050f5d.pdf
- Whole digital version of the book may still be available

Admin

- Mistakes in slides
  - They might occur! Don’t be afraid to correct me … if you are >99% sure I am wrong
- Lab for online students
  - Might be difficult to deal with questions real time (we will try)
  - We will check with you regularly and specifically with 30 minutes to go of each lab and try to handle questions

Outline

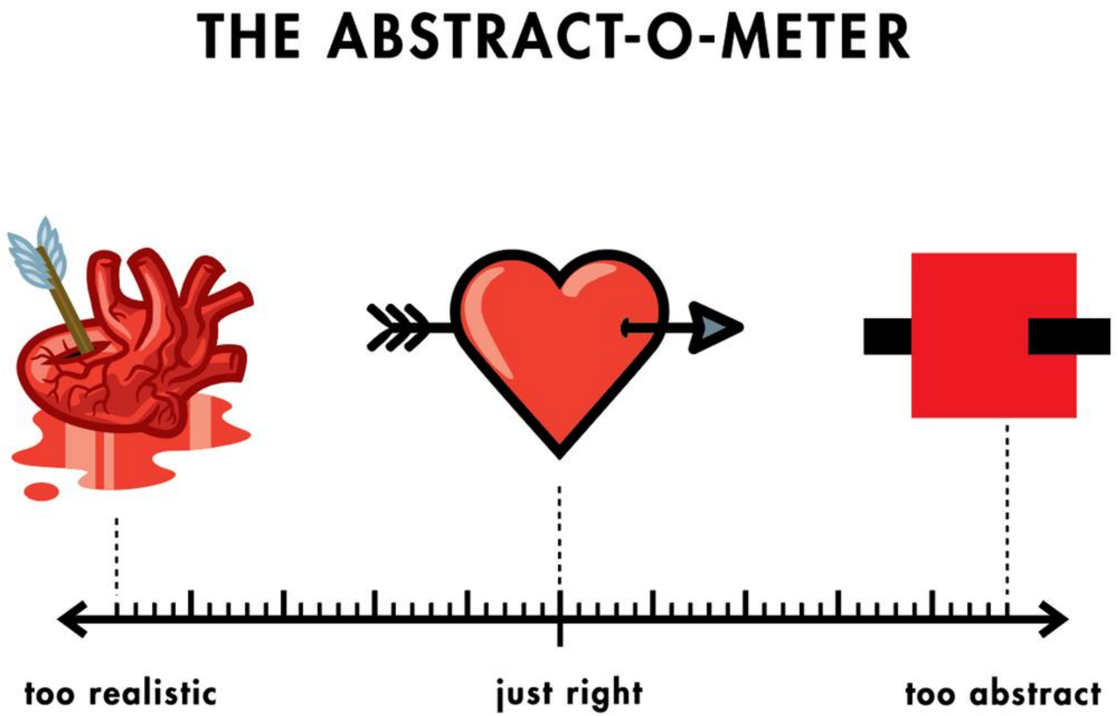
- Boolean logic
- Boolean function synthesis
- Hardware description language (HDL)
- Hardware simulation
- Multi-bit buses

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

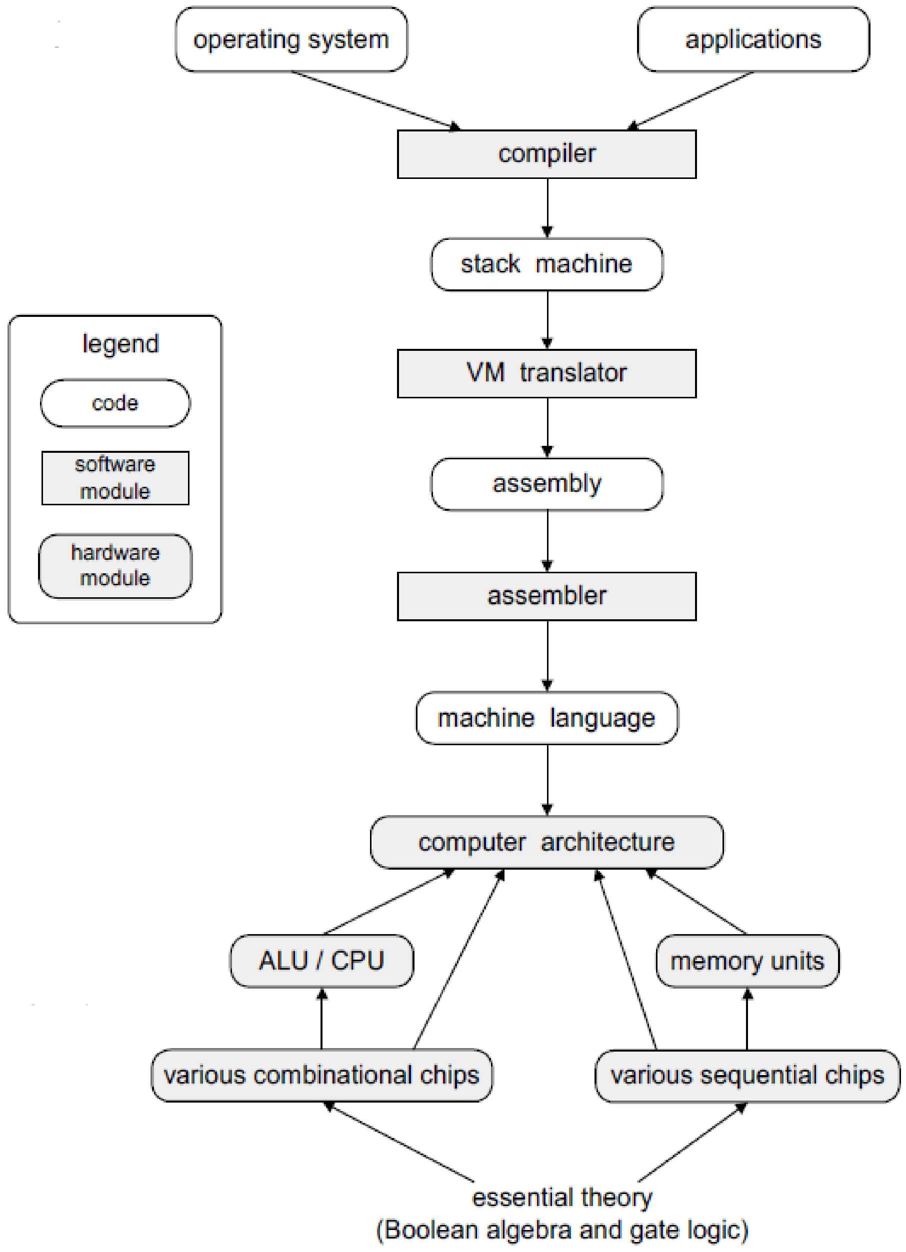
Abstractions

- 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



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



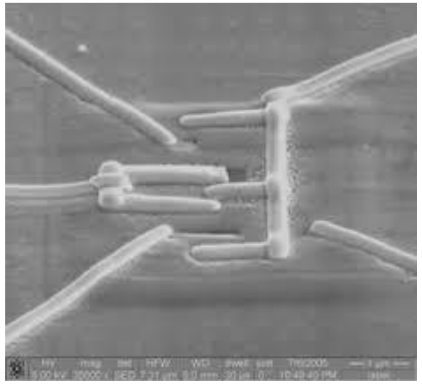
Top-down vs Bottom-up

- 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

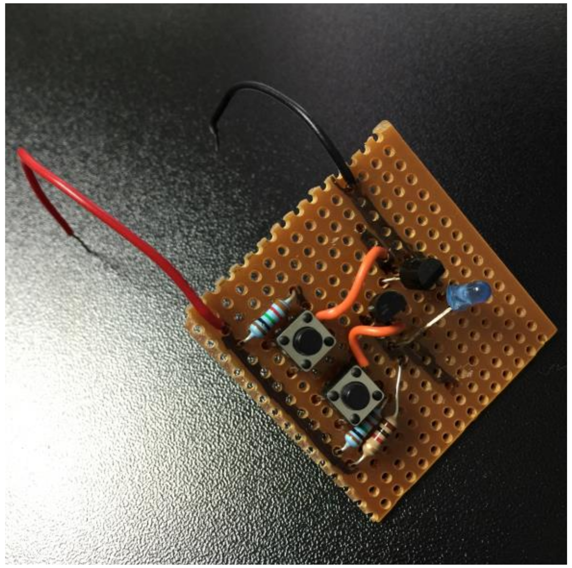
Building from Elementary Components

- Components inside a computer is build up of billions of nano sized transistors



Nano size transistor

- The transistor is a basic building block used to construct more complex electronic components

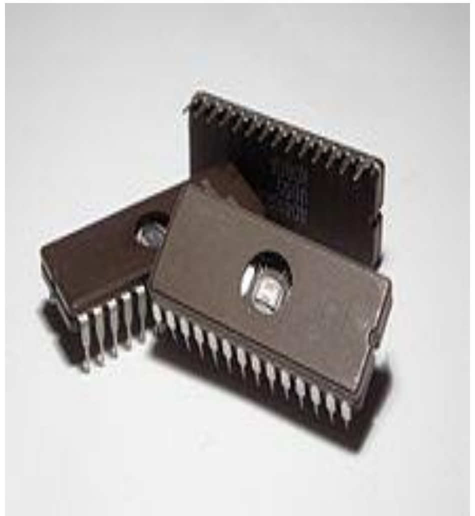


Transistors on a Circuits Board With other electronic components connected as a NAND Gate

Building from Elementary Components

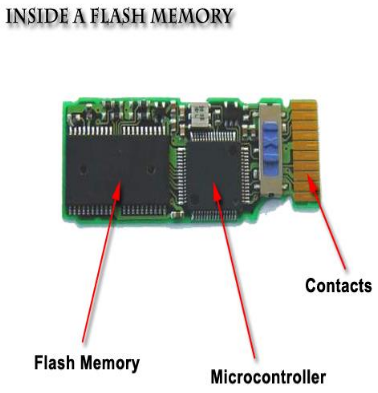
- The CPU, RAM, ROM are built up from nano sized transistors

Central Processing Unit (CPU)



Erasable Programmable Read Only Memory (EPROM)

Random Access Memory (RAM)



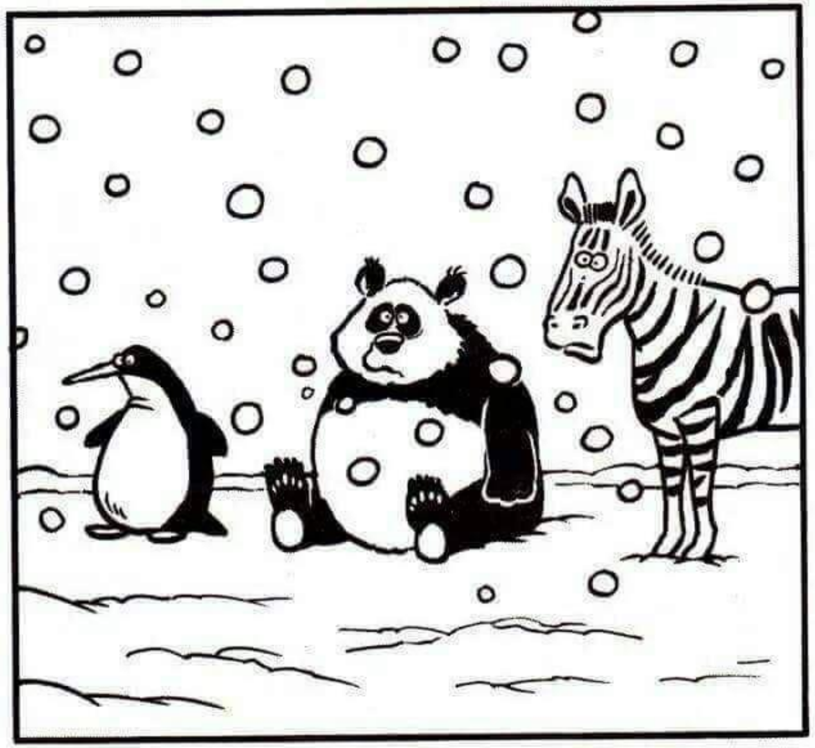
Flash Memory

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

~ John Ashbery (b.1972), American Poet

Boolean (Binary) Values

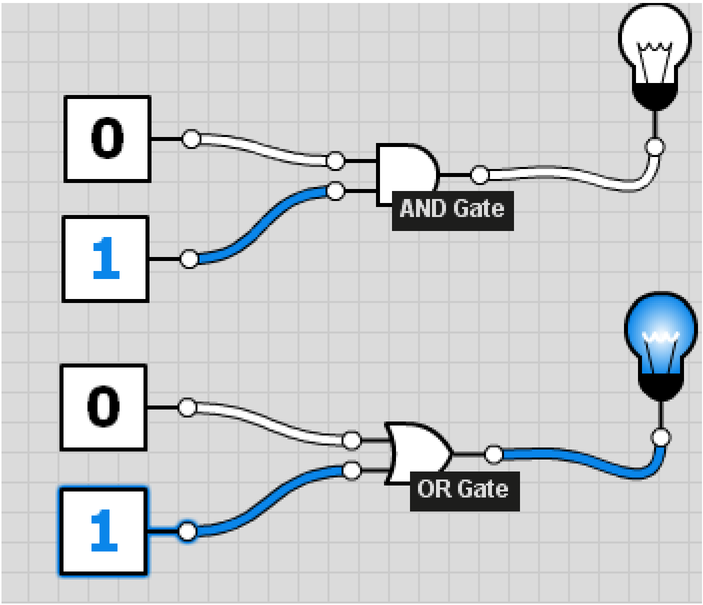
Off	On
False	True
N	Y
0	1
Black	White



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



https://upload.wikimedia.org/wikipedia/commons/2/2a/LogicGatesWorking.png

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
  - NAND
  - We will see this later
- **Therefore every possible chip can be built from just the NAND gates**

George Boole, 1815-1864

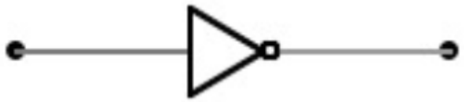
("*A Calculus of Logic*")

Boolean Function

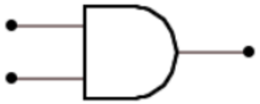
- A Boolean function is a function that operates on binary inputs and return binary outputs
- Truth table is **every possible function evaluation** of the input variables
- [note 0 and 1 used to define false and true]
- Everything can be defined by a truth table

Elementary Logic Gates

$A = \bar{A}$

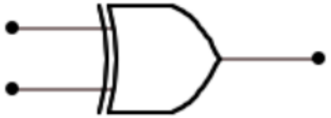


$A\ AND\ B = A \cdot B$

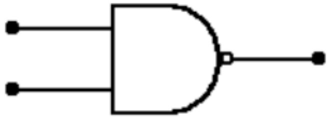


$A\ OR\ B = A + B$

$A\ XOR\ B = A \oplus B$



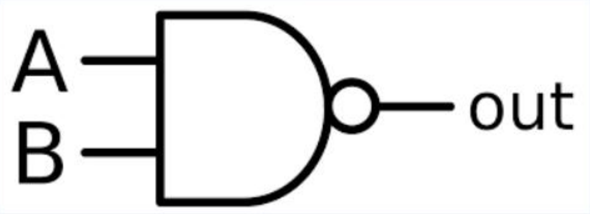
$A\ NAND\ B = \overline{A \cdot B}$



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

Representations of Simple Logic Gate

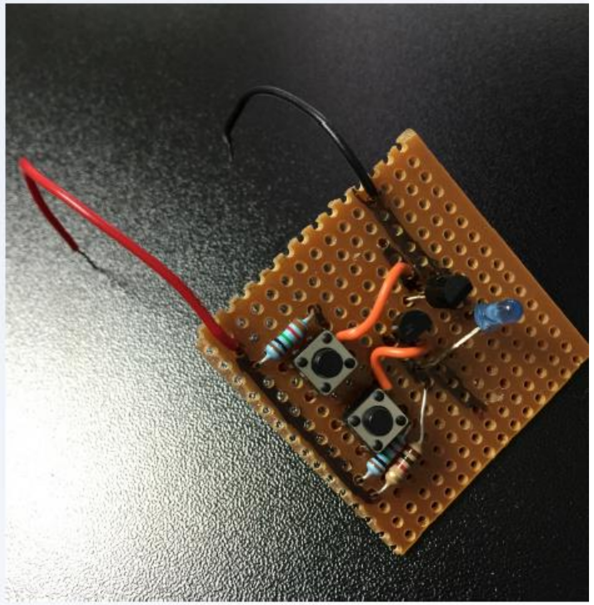
They are all the same!



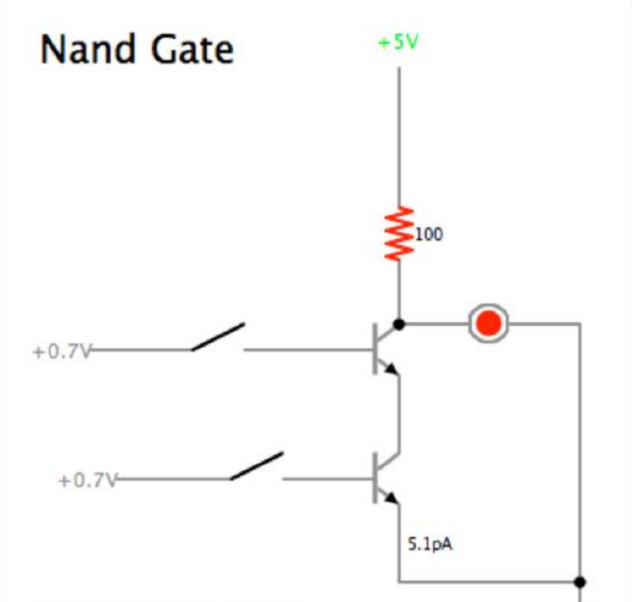
Logic Gate Diagram

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.

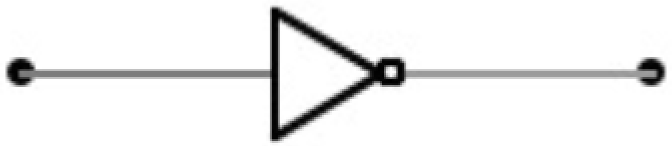


Circuit Diagram

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

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.



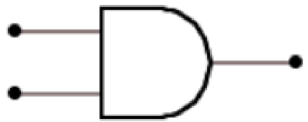
A	$\bar{A}$
0	1

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

AND gate

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

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



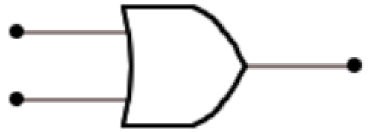
A	B	$A \bullet B$
0	0	0
0	1	0
1	0	0
1	1	1

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

OR gate

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

$$A\ OR\ B = A + B$$



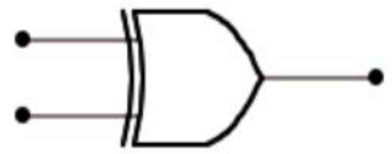
A	B	$A+B$
0	0	0
0	1	1
1	0	1
1	1	1

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

XOR gate

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

$$A\ XOR\ B = A \oplus B$$



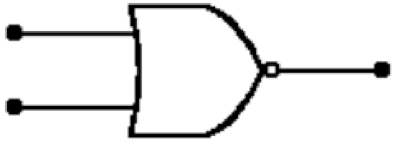
A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

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

NOR gate

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

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



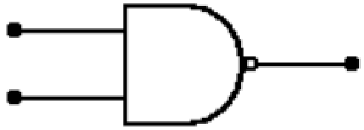
A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

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

NAND gate

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

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



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

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

Collection of Elementary Logic Gates

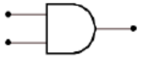
$A = \bar{A}$

A	Ā
0	1

$A\ XOR\ B = A \oplus B$

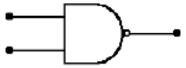
A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

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



A	B	$A \bullet B$
0	0	0
0	1	0
1	0	0
1	1	1

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



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

$A\ OR\ B = A + B$

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

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

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)

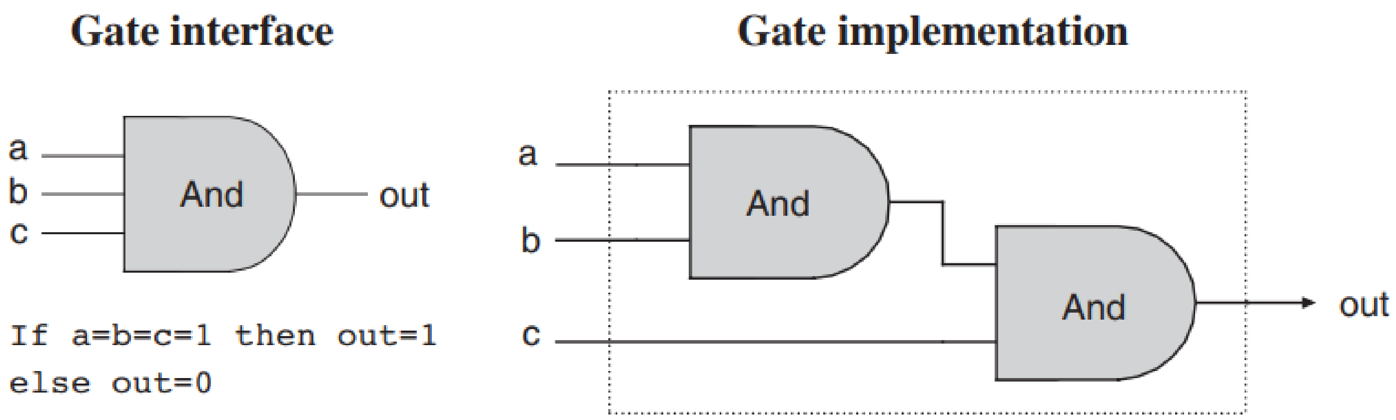


Figure 1.4 Composite implementation of a three-way And gate. The rectangle on the right defines the conceptual boundaries of the gate interface.

Composite Gates

- What gate is the truth table on the right?

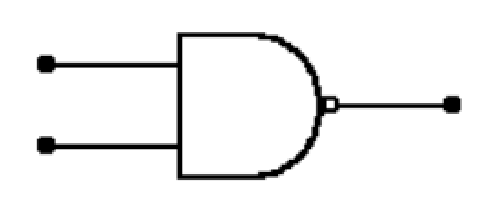
$$f(A,B) = \overline{A \bullet B}$$

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!

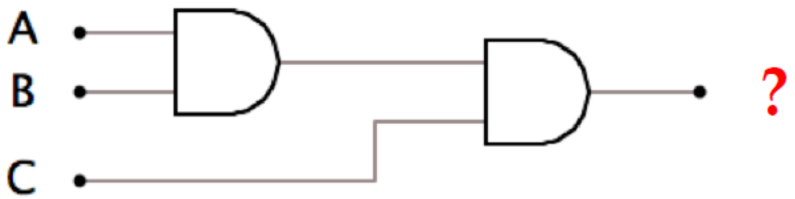


A	B	$\overline{A \bullet B}$
0	0	1
0	1	1
1	0	1
1	1	0

Composite Gates

Three Input And Gates

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

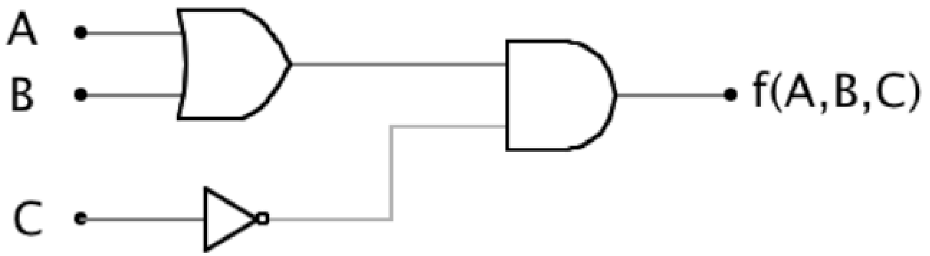


A	B	C	*f*(A,B,C)
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Composite Gates

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

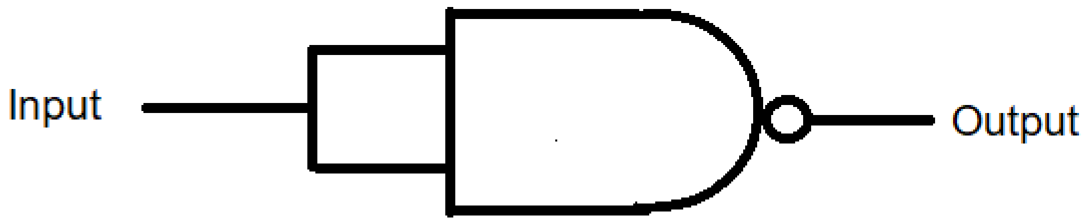
(A OR B) AND NOT C



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	

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

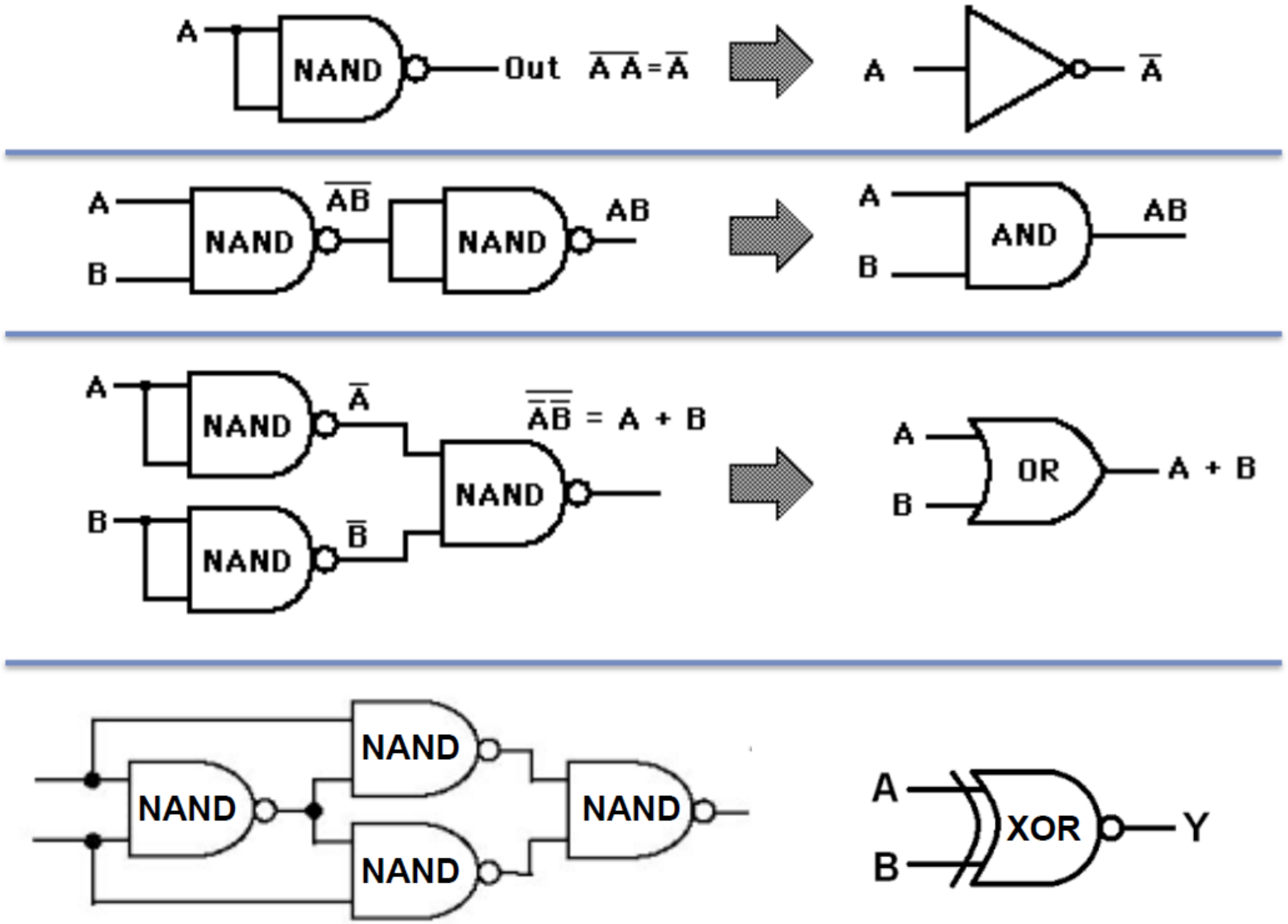
The Super NAND

- 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 Or y = Not(Not(x) And Not(y))
- All Boolean logic can be built using Nand gates
  - Because….
  - Not(x) = Nand(x,x)
  - And(x,y) = Not(Nand(x,y))
  - = (Nand(Nand(x,y), 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) = Not(a) Or(b) And(c)

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

Boolean Functions

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

Boolean Functions

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

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 And y) Or (Not(x) 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 And 0) Or (Not(0) And 1) =
0 Or (1 And 1) =
0 Or 1 = 1

(1 And 1) Or (Not(1) And 0) =
1 Or (0 And 0) =
1 Or 0 = …..

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 Or x=x   x And x=x   x Or x Or x Or x ….. = x**
  - Operation can be applied multiple times without changing the result beyond the initial application

And is implicit

- **Associative law: (x Or y)Or z = x Or(y Or z)   (x And y)z = x(y 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)

Laws of Boolean Algebra

- **Commutative law: x And y = y And x   x Or y=y Or x**
  - Function outcome is unaltered by reordering its terms

- **Distributive law: x and (y or z) = (x and y) or (x and z)**

  x or (y and z) = (x or y) and (x or z)

same as normal math when you expand out an equation

Laws of Boolean Algebra

- **De Morgans Law:**
  - **Not(x And y) = Not(x) Or Not(y)** *[1]*
  - **Not(x Or y) = Not(x) And 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 And(Not(x))=0   x Or (Not(x))=1**
  - A term And with its complement equals 0 and a term Or with its complement equals 1
- **Double negative Law: (NOT(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	
$A = A$	$\overline{A} = \overline{A}$
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	
$\overline{\overline{A}} = A$	
6. Complementary Law	
$A \cdot \overline{A} = 0$	$A + \overline{A} = 1$
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	
$A \cdot (\overline{A} + B) = AB$	$A + (\overline{A} \cdot B) = A + B$
12. De Morgan's Law	
$\overline{A \cdot B} = \overline{A} + \overline{B}$	$\overline{A + B} = \overline{A} \cdot \overline{B}$

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

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

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

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

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

Boolean Algebra

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

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

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

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

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

Boolean Algebra

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

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

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

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

De Morgan law (can use both ways around)

Not(Not(x Or y))=

Simplify Boolean Expression

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

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

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

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

Not(Not(x Or y))=

double negation

x 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 And y →

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

→ x And y

From 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

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?

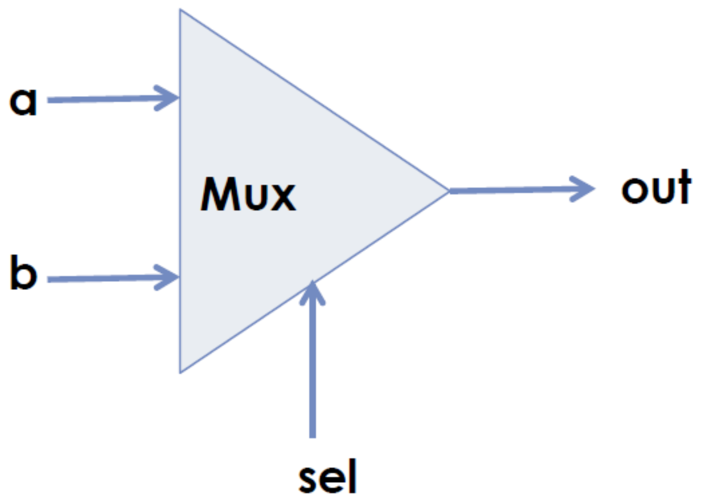
"everything should be as simple as it can be but not simpler"
Einstein

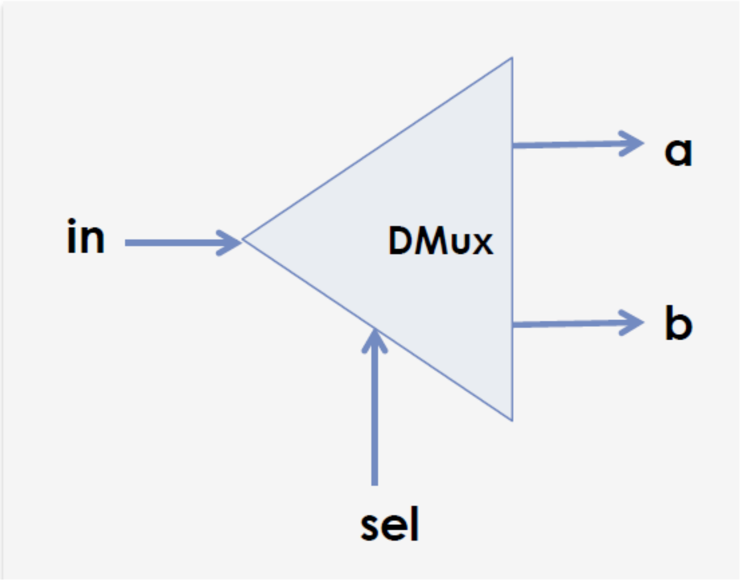
- (Not(x) And Not(y) And Not(z)) Or
  (Not(x) And y And Not(z)) Or
  (x And Not(y) And Not(z))
- (Not(x) Or (x And Not(y))) And Not(z)
  - Both correct

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

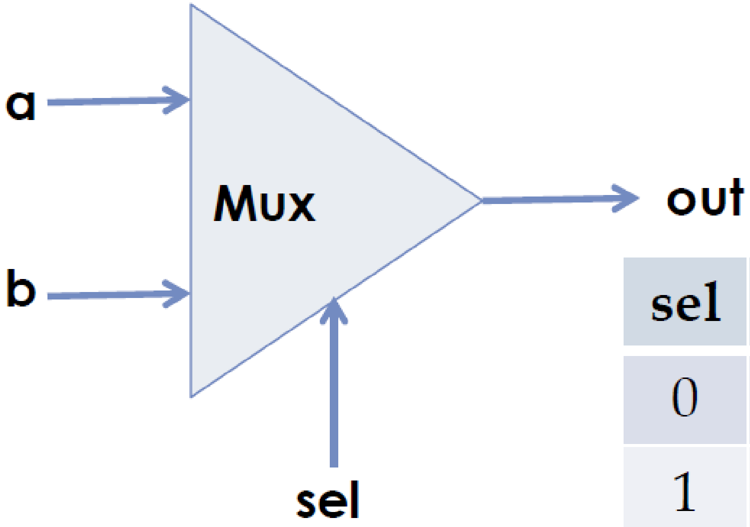
- A multiplexer is a combinational circuit that provides **single output** but accepts **multiple data inputs**.
- A demultiplexer is a combinational circuit that takes **single input** but that input can be directed through **multiple outputs**.





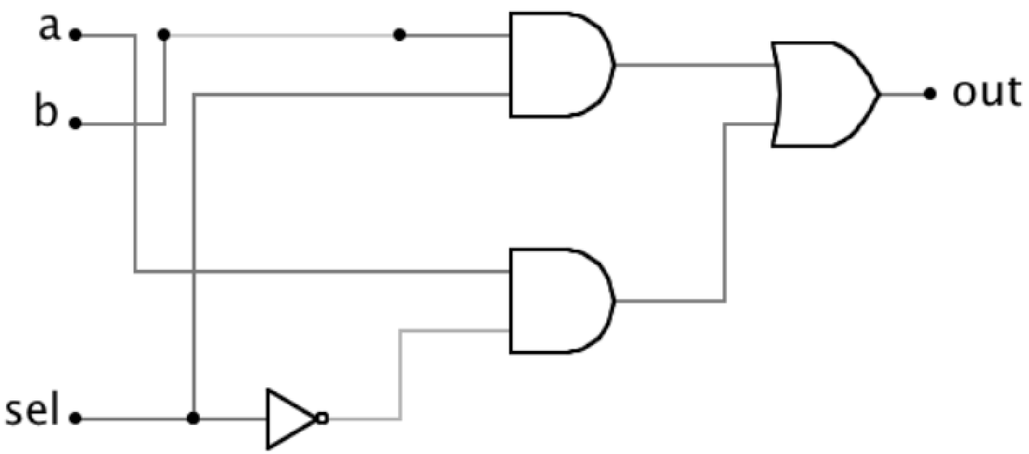
Multiplexer

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



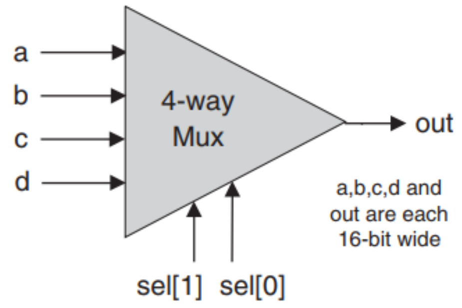
sel	out
0	a
1	b

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

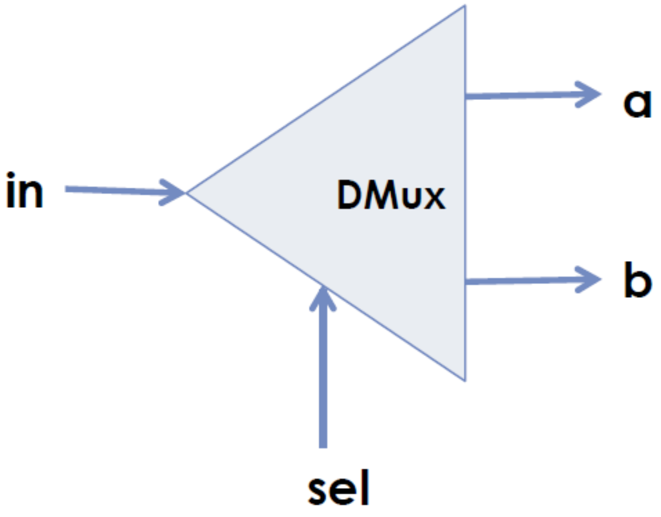


Demultiplexer

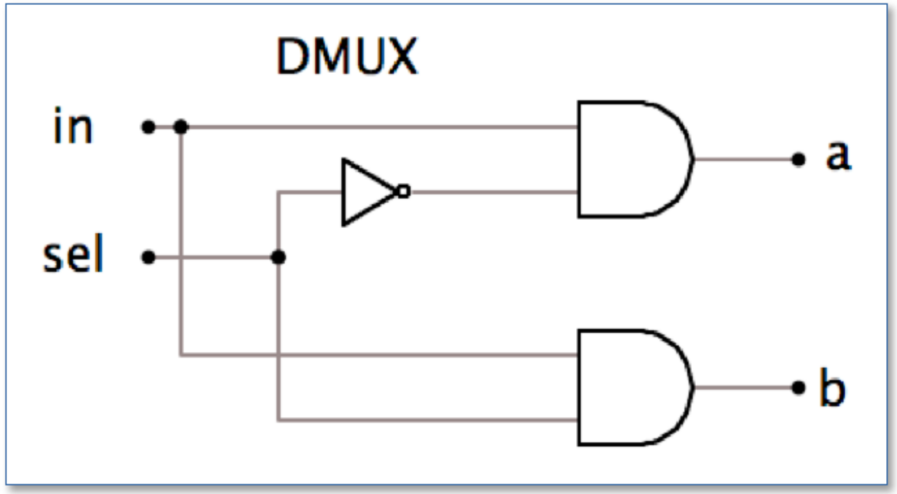
sel[1]	sel[0]	out
0	0	a
0	1	b
1	0	c
1	1	d



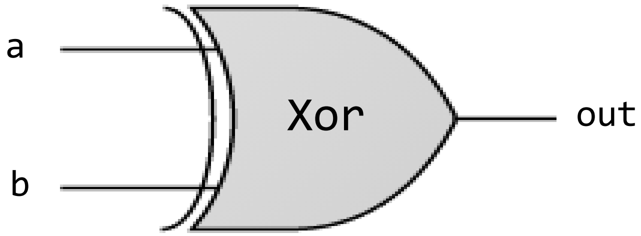
sel	a	b
0	in	0
1	0	in



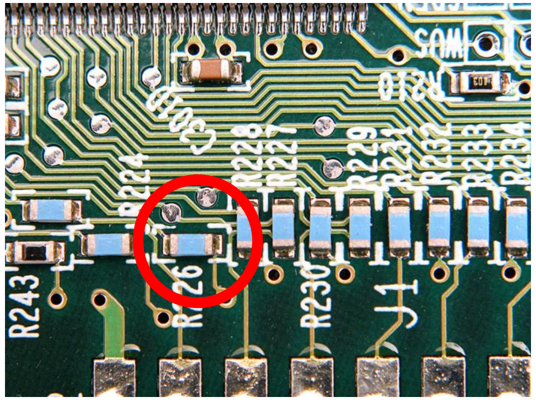
If sel = 0 then
{a = in; b = 0}
else
{a = 0; b = in}



Building a logic gate



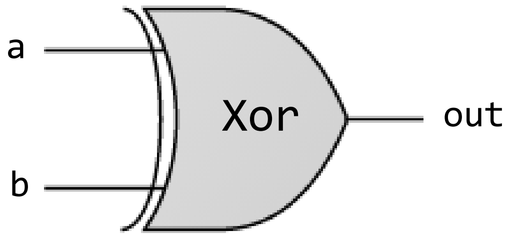
outputs 1 if one, and only one, of its inputs, is 1.



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**

<u>Requirement:</u>

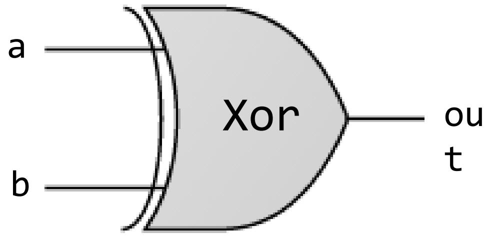
Build a gate that delivers this functionality

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

CHIP Xor {
    IN a, b;
    OUT out;

    PARTS:
    // Implementation missing
}
```

<u>Gate interface</u>

Expressed as an HDL stub file

Requirements to gate diagram



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**

<u>Requirement:</u>

Build a gate that delivers this functionality

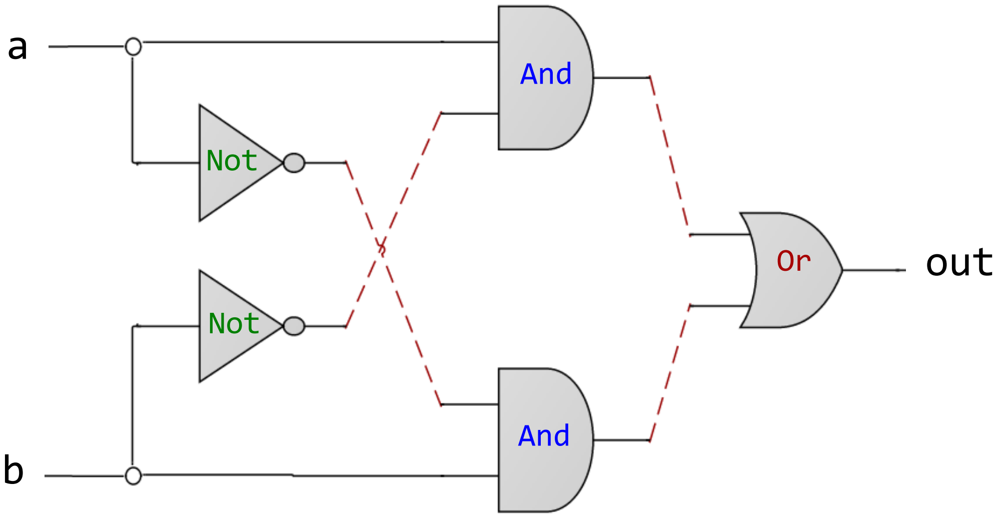
<u>General idea:</u>

out=1 when:

a And Not(b)

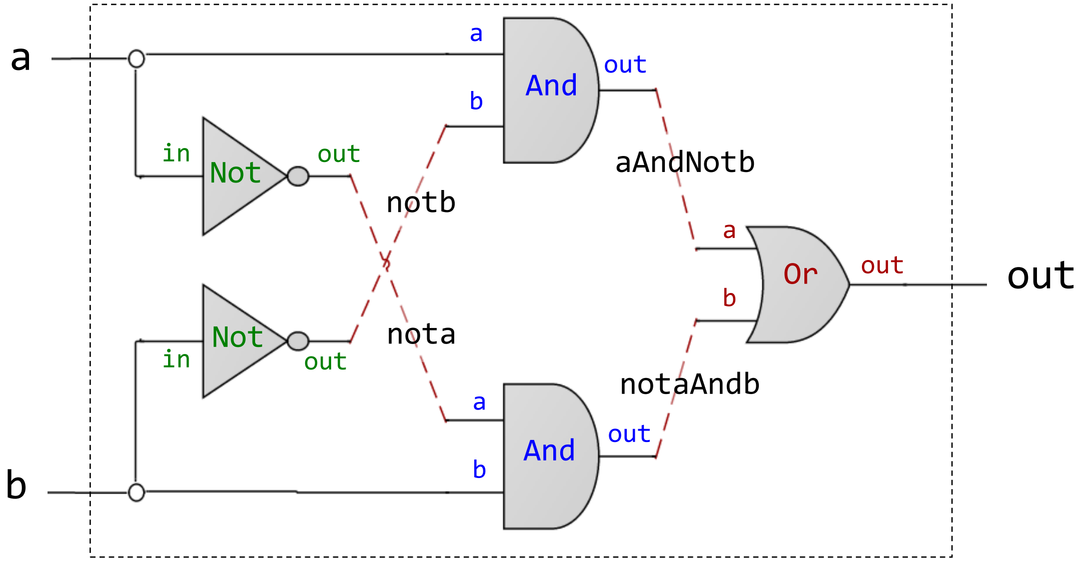
Or

b And Not(a)



!Precendence

Gate Diagram to HDL Code



interface

implementation

```
/** Xor gate: 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 Simulator

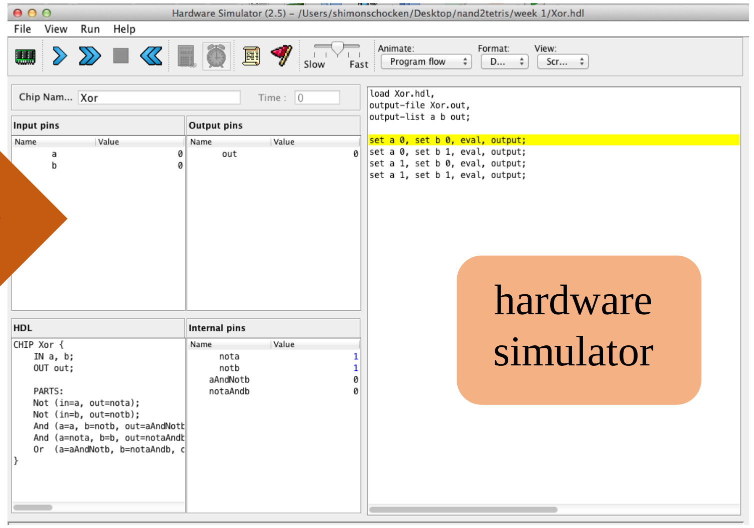
HDL code

```
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 (
    Or
}
```

test script

```
load Xor.hdl,
output-file And.out,
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;
```

simulate



2 Options

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

Simulation options:

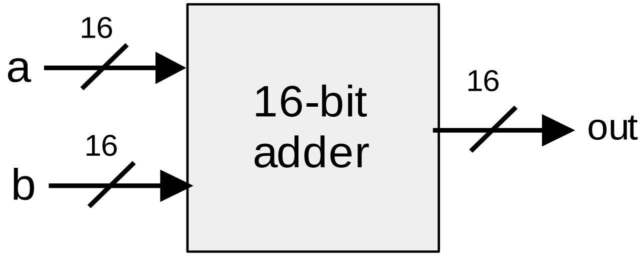
- Interactive
- Script-based
- With / without output and compare files

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], temp=a[1], out=t01);
   And(a=t01,  temp=a[2], out=t012);
   And(a=t012, temp=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