

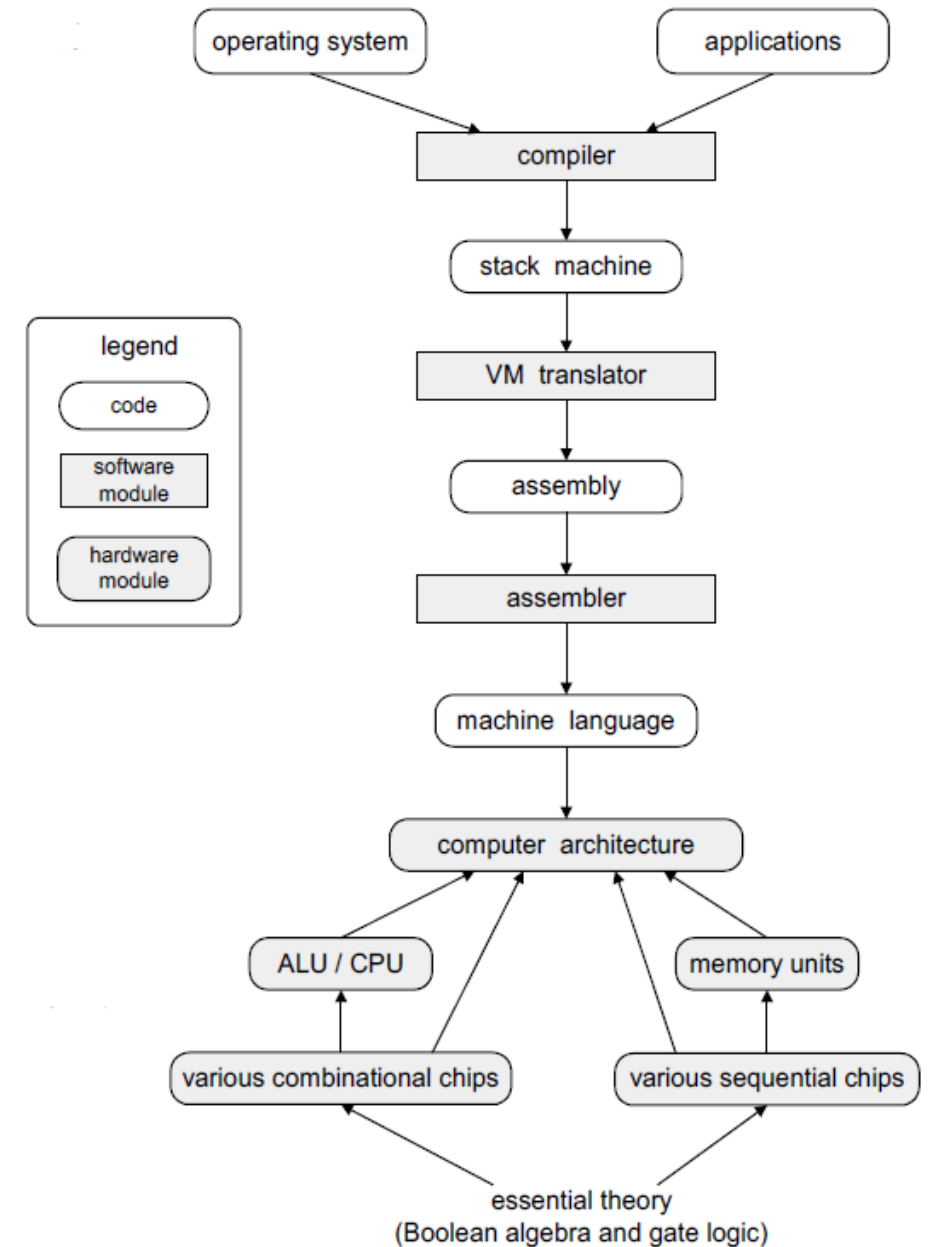
Lecture 2 – Boolean Logic

11/10/2024

Dr. Libin HONG

Preamble

- We should all be able to run the Nand2Tetris hardware simulator
- We should all now be able to edit and test .hdl files
- <https://www.nand2tetris.org/>

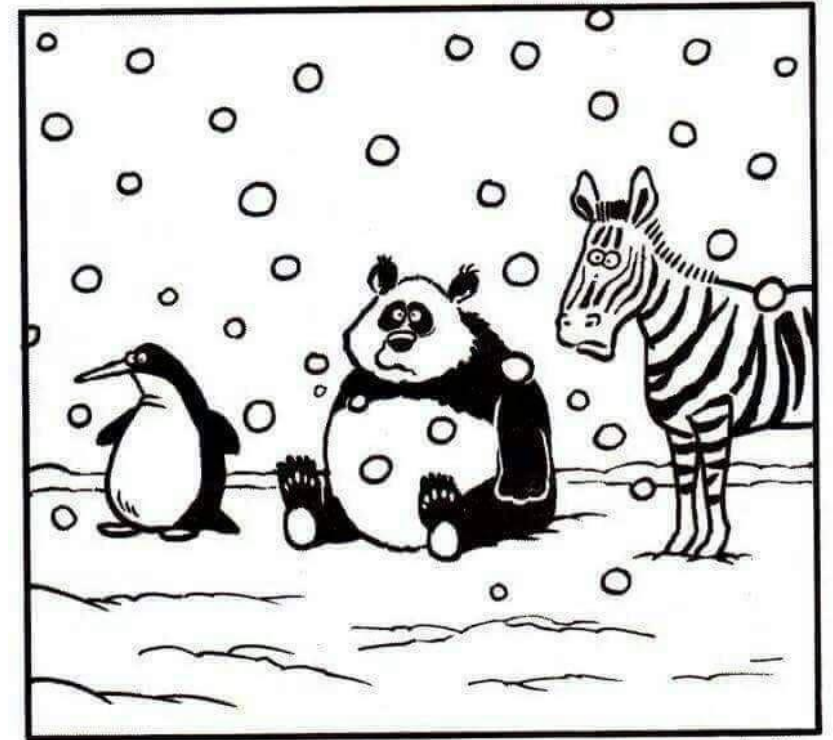


A few common mistakes

- HDL files need simple characters (% \$ _ etc not allowed)
- Save as xor1.hdl
 - Not xor1.hdl.txt (.txt is sometimes be added a default by notepad)
- Software has inbuilt versions of basic (eg. And) chips but...
- ..will use your version of the chip if it exists...
- ..if your version is wrong then higher level chips wont work)

Lecture 2 - Boolean Logic

- 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, sub-buses



Boolean Colouring picture for lazy people.

Boolean Values



False

N

0

Black

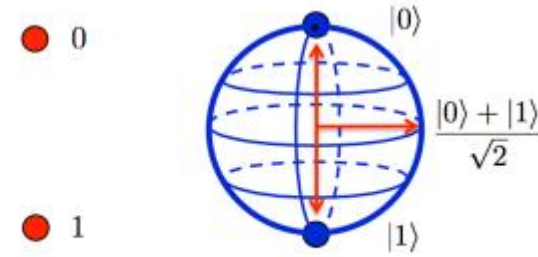


True

Y

1

White



Classical Bit

Qubit

<https://chriskohlhepp.wordpress.com/economics-robotics/experimenting-with-quantum-lisp-lambda-calculus-and-quantum-physics/>



Boolean Logic

All chips constructed from elementary logic gates

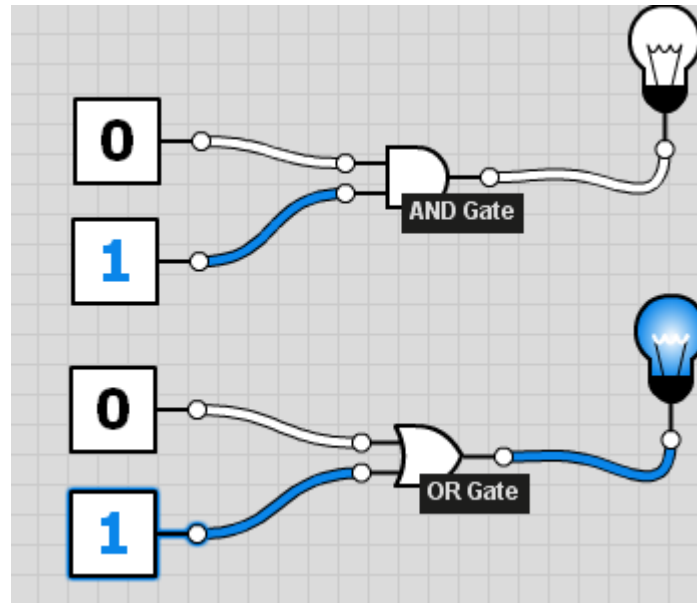
- Every chip can be built from a combination of:
 - AND (eg. If x and y are true then z is true, otherwise false)
 - OR (eg. If x or y are true then z is true, otherwise false)
 - NOT (eg. If x is true the z is false)
 - No integration, division, differentiation...
 - “canonical representation”We will prove this later
- AND, OR and NOT can be built from NAND
 - NAND (eg. If x and y are true then z is false, otherwise true)
 - We will prove this later
- Therefore every possible chip can be built from just the NAND gate!!!!
- (reductive and elegant but seldom optimal)
- [more later....]



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

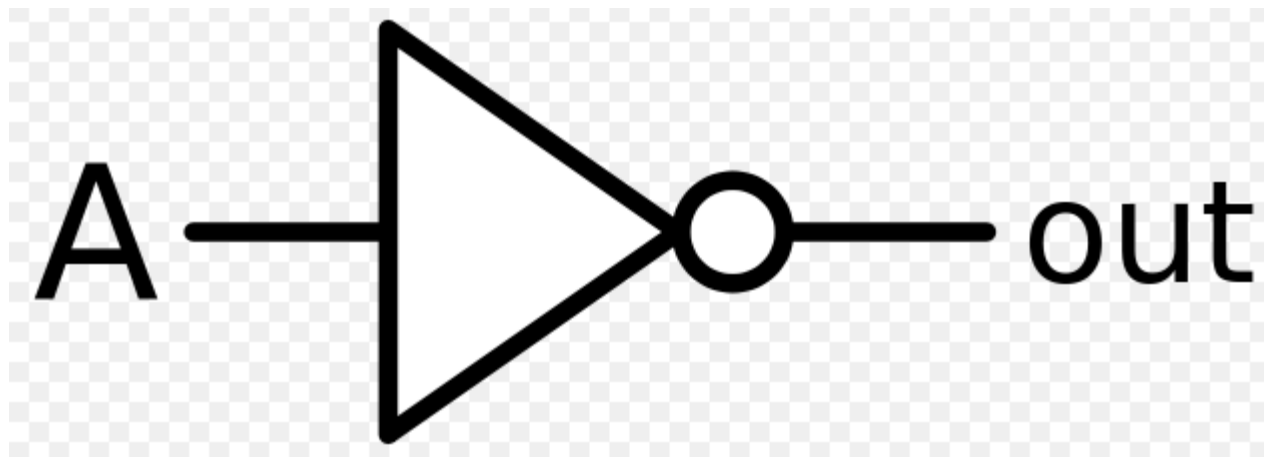
Electronics

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



Not gates

- NOT gate – inverter – if 0 then 1
- (important, only uses one entry)
- The “bubble” (o) at the end of the NOT gate symbol denotes a signal inversion (complementation) of the output signal.
- This bubble can also be present at the gates input to indicate an active-LOW input. (usually active high by default)



Boolean Function

- A Boolean function of the form $f(v_1, \dots, v_n)$ where v and f can only take the values of 1 or 0
- Several different ways to represent this

x And y

$$x \wedge y$$

x Or y

$$x \vee y$$

Not(x)

$$\neg x$$

$$x'$$

– Not(x) : \bar{x}

– And(x,y) : $x \cdot y$ or xy

– or(x,y) : $x + y$

Truth Tables

• $x, y \mid f(x,y)$

• $0, 0 \mid 0$

• $0, 1 \mid 1$

• $1, 0 \mid 1$

• $1, 1 \mid 1$

• ?

• OR

$x, y \mid f(x,y)$

$0, 0 \mid 0$

$0, 1 \mid 0$

$1, 0 \mid 0$

$1, 1 \mid 1$

AND

$x \mid f(x)$

$0 \mid 1$

$1 \mid 0$

NOT

$x, y \mid f(x,y)$

$0, 0 \mid 1$

$0, 1 \mid 1$

$1, 0 \mid 1$

$1, 1 \mid 0$

NAND

- Truth table is every possible function evaluation of the input variables
- [note 0 and 1 used to define false and true]

Truth Tables

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

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

x	Not
0	1
1	0

All possible Boolean functions of 2 variables

Function	x	0	0	1	1
	y	0	1	0	1
Constant 0		0	0	0	0
And		0	0	0	1
x And Not(y)		0	0	1	0
x		0	0	1	1
Not(x) And y		0	1	0	0
y		0	1	0	1
Xor		0	1	1	0
Or		0	1	1	1
Nor		1	0	0	0
Equivalence		1	0	0	1
Not y		1	0	1	0
if y then x		1	0	1	1
Not x		1	1	0	0
If x then y		1	1	0	1
Nand		1	1	1	0
Constant 1		1	1	1	1

Boolean Expressions

Not(0 Or (1 And 1)) =

Not(0 Or 1) =

Not(1) =

0

Not(0) Or (1 And 1) =

Not(0) Or (1) =

1 Or 1 =

1

**(Brackets
Matter)**

PRECEDENCE

- PRECEDENCE

Parentheses evaluated first

Not binds most tightly

Then **And**

Then **Or**

Mathematicians may argue that precedence is for the lazy

Boolean Expression – using precedence

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

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

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

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

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

$$x = 0$$

Boolean Functions

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

Boolean Functions

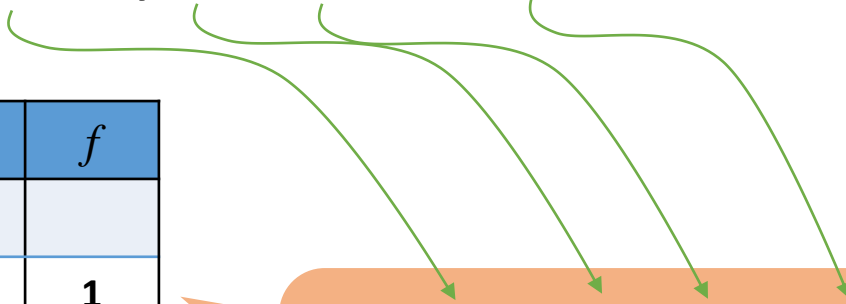
$$f(x, y, z) = (x \text{ And } y) \text{ Or } (\text{Not}(x) \text{ And } z)$$

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	



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

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

Boolean Functions

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

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

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

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

Boolean Functions

$$f(x, y, z) = (x \text{ And } y) \text{ Or } (\text{Not}(x) \text{ And } z) \quad \left. \vphantom{f(x, y, z)} \right\} \text{ formula}$$

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

Boolean function simplification rules

- **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:** $(x \text{ Or } y) \text{ Or } z = x \text{ Or } (y \text{ Or } z)$ $(x \text{ And } y) \text{ And } z = x \text{ And } (y \text{ And } z)$
 - Terms may be associated in any way desired if same logical operations
- **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:** $x \text{ and } (y \text{ or } z) = (x \text{ and } y) \text{ or } (x \text{ and } z)$
 $x \text{ or } (y \text{ and } z) = (x \text{ or } y) \text{ and } (x \text{ or } z)$

Boolean function simplification rules

- **Complement Law: $x \text{ And } (\text{Not}(x))=0$ $x \text{ Or } (\text{Not}(x))=1$**
 - A term And'ed with its complement equals 0 and a term Or'ed with its complement equals "1"
- **Involution Law: $(\text{NOT}(\text{NOT}(x)))=x$**
 - A function that, when applied twice, brings one back to the starting point.
(double negation)
- **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]	[2]
1	1
1	0
1	0
0	0

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)) =

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

Not((Not(x) And Not(x)) And 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)

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

Boolean Algebra

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

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

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

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

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

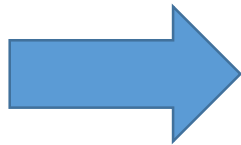
double negation

$x \text{ Or } y$

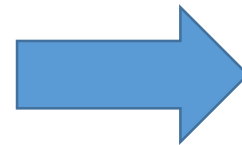
Boolean function synthesis

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

$x \text{ And } y$



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



$x \text{ And } y$

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

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

Or

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

Or

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

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

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

Complement Law

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

Distributive law

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

Why simplify?

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

Every chip **can** be built with NAND gates

- All Boolean logic can be built from And, Or and Not
 - Because....
 - 'Truth table to Boolean expression' method just uses these three operations
- 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....
 - $\text{Not}(x) = \text{Nand}(x, x)$
 - $\text{And}(x, y) = \text{Not}(\text{Nand}(x, y))$

x	y
0	0
0	1
1	0
1	1

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)

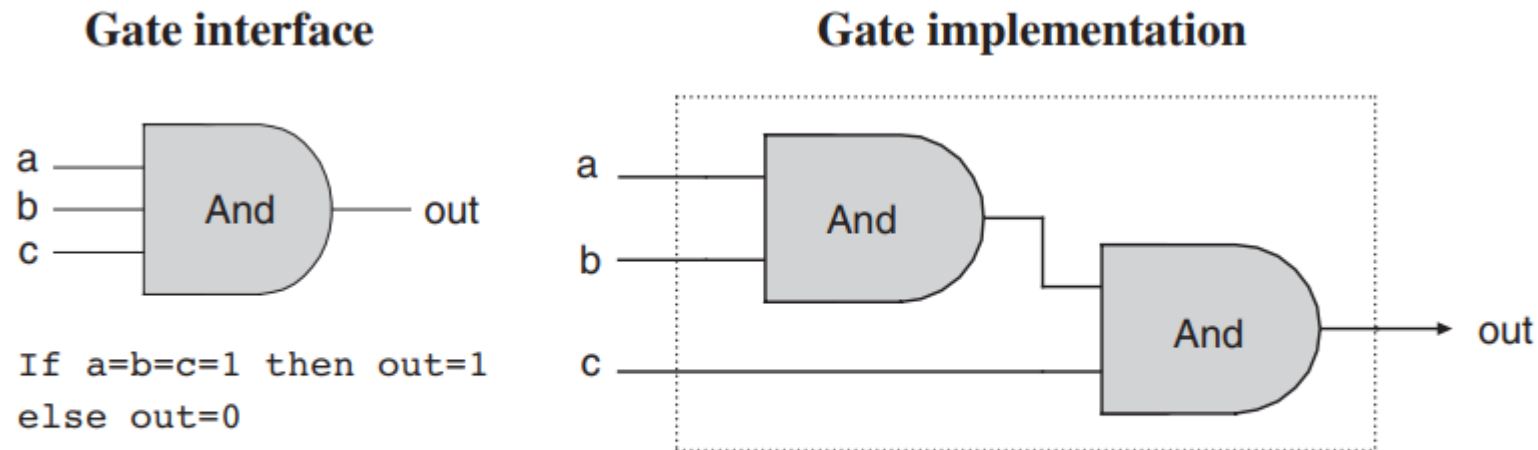
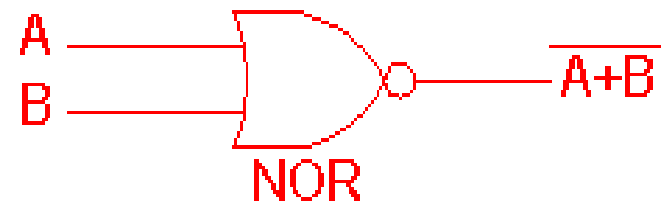
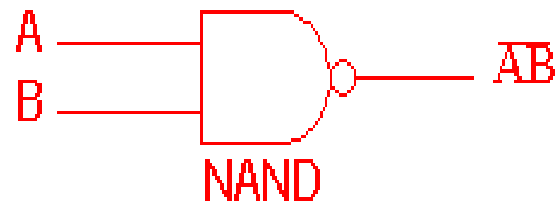
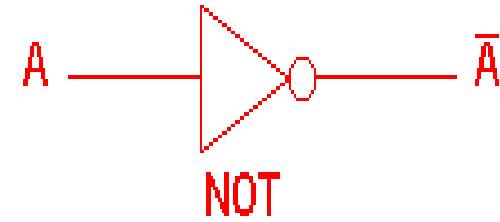
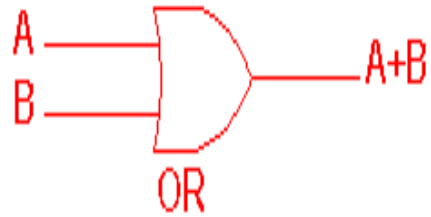
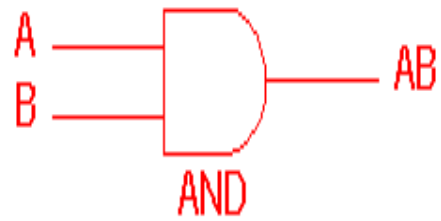
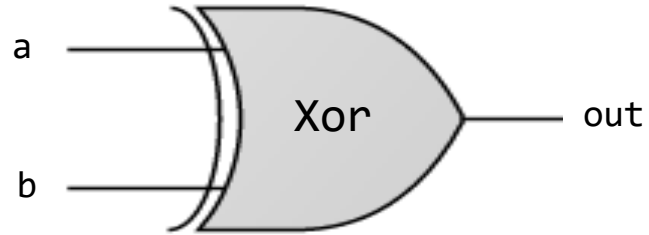


Figure 1.4 Composite implementation of a three-way And gate. The rectangle on the right defines the conceptual boundaries of the gate interface.

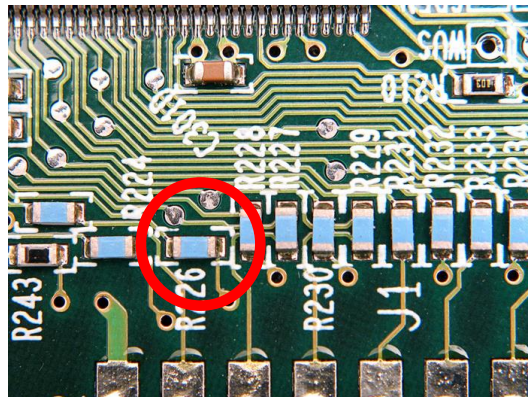
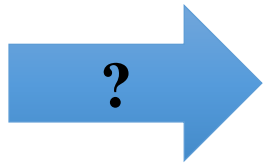
Logic Gates



Building a logic gate



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

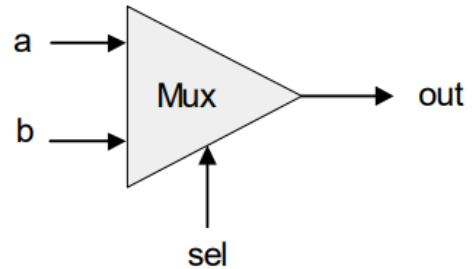


The Process:

- ✓ 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.

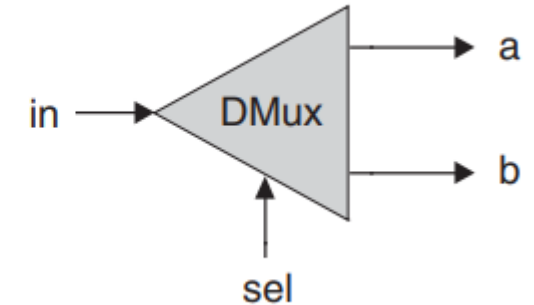
Multiplexers and Demultiplexers

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

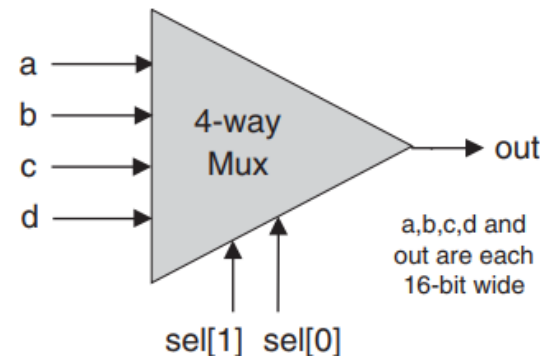


sel	out
0	a
1	b

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



XOR -aside

- XOR Truth table

- $x, y \mid f(x,y)$

- $0, 0 \mid 0$

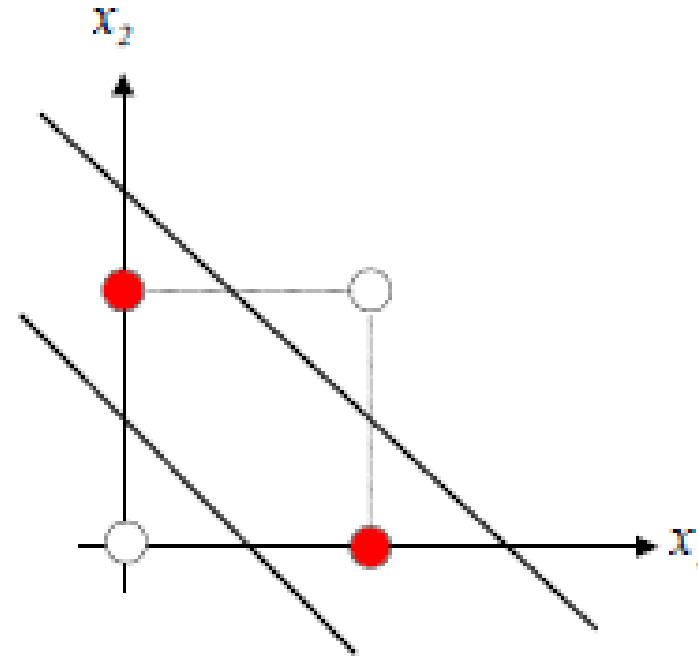
- $0, 1 \mid 1$

- $1, 0 \mid 1$

- $1, 1 \mid 0$

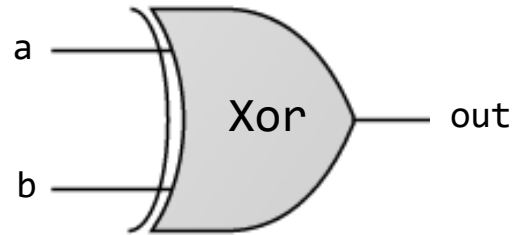
- Non-linear

- Roadknight, Chris, et al. "Supervised learning and anti-learning of colorectal cancer classes and survival rates from cellular biology parameters." *Systems, Man, and Cybernetics (SMC), 2012 IEEE International Conference on*. IEEE, 2012.



<https://tex.stackexchange.com/questions/140741/draw-graph-of-xor-problem-neural-network?rq=1>

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:
```

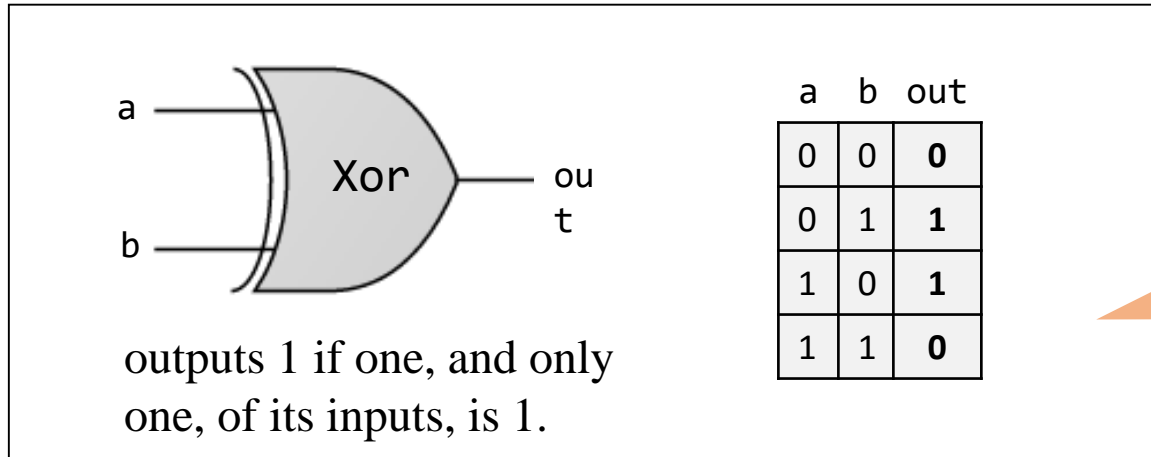
```
    // Implementation missing
```

```
}
```

Gate interface

Expressed as an HDL stub file

Requirements to gate diagram



Requirement:

Build a gate that delivers this functionality



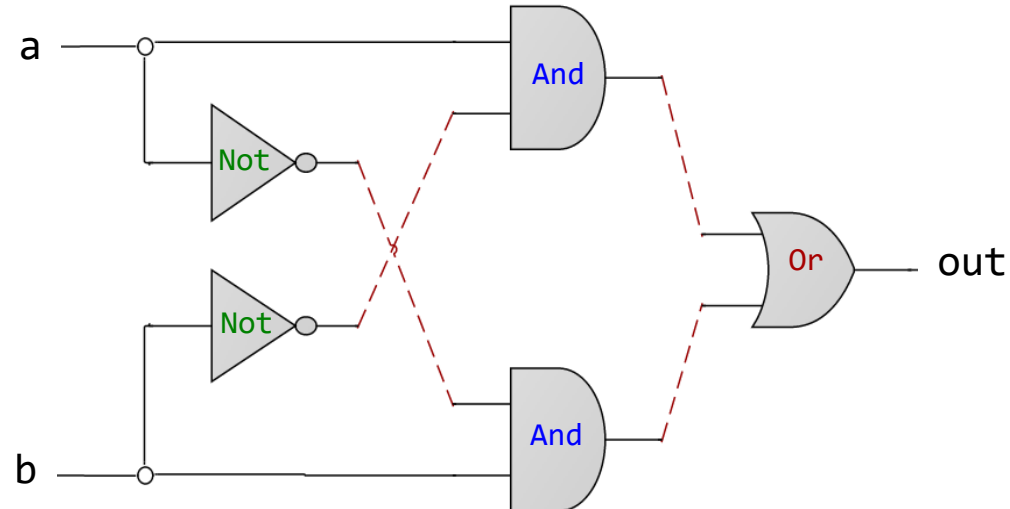
General idea:

out=1 when:

a And Not(b)

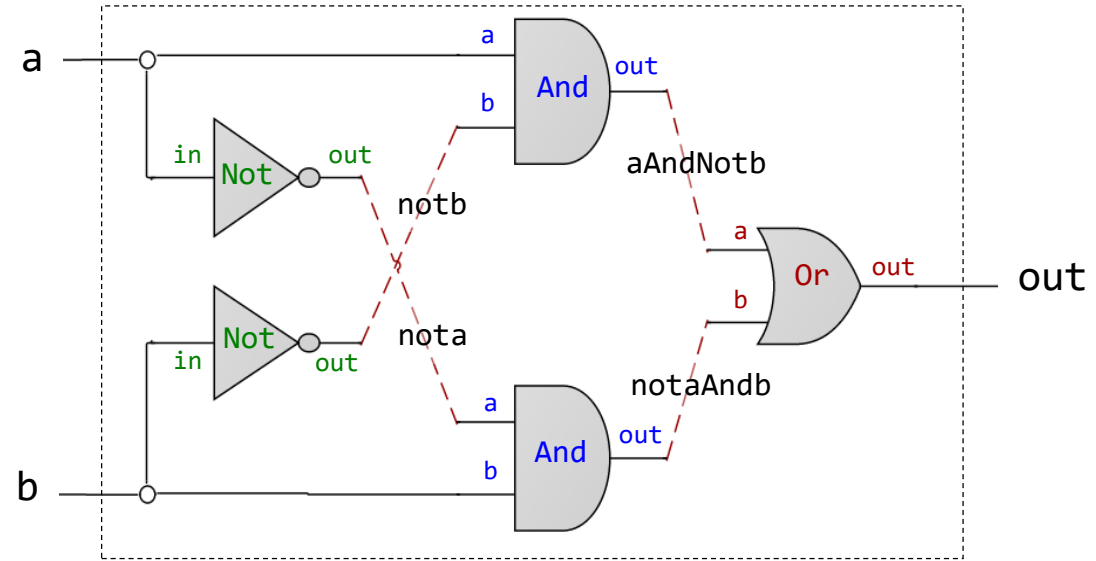
Or

b And Not(a)



!Precedence

Gate Diagram to HDL Code



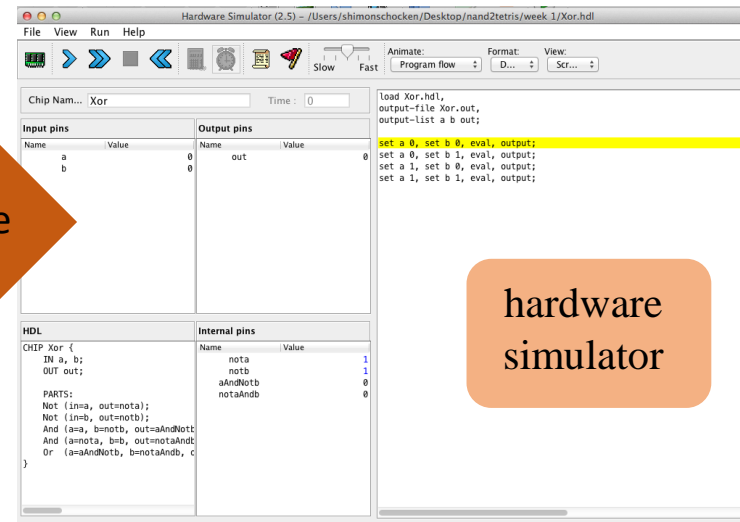
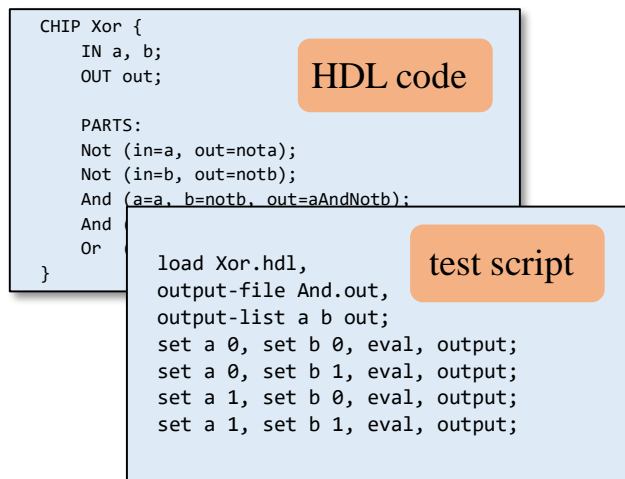
interface

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



2 Options

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

Simulation options:

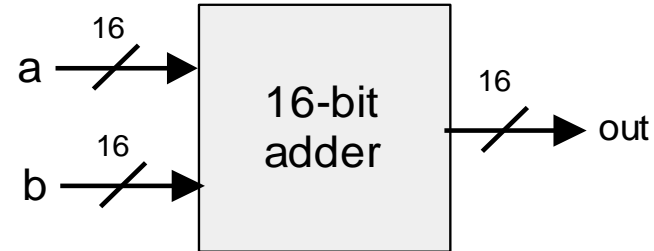
- Interactive
- Script-based
- With / without output and compare files

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

```
/*
 * 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

Sub-Buses

Buses can be composed from (and decomposed into) sub-buses

```
...  
IN lsb[8], msb[8], ...  
...  
Add16(a[0..7]=lsb, a[8..15]=msb, b=..., out=...);  
Add16(..., out[0..3]=t1, out[4..15]=t2);
```

Some syntactic choices of the Nand2Testris HDL

- buses are indexed right to left: if $a[16]$ is a 16-bit bus, then $a[0]$ the right-most bit, and $a[15]$ is the left-most bit
- overlaps of sub-buses are allowed in output buses of parts
- width of internal pin buses is deduced automatically
- The **false** and **true** constants may be used as buses of any width.



Questions?????

