

## Chapter 1

# Boolean Logic

These slides support chapter 1 of the book

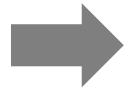
*The Elements of Computing Systems*

By Noam Nisan and Shimon Schocken

MIT Press

# Chapter 1: Boolean logic

---



## Boolean logic

- Boolean function synthesis
- Hardware description language
- Hardware simulation
- Multi-bit buses
- Project 1 overview

# Boolean Values



F

T

N

Y

0

1

# Boolean Operations

$x$  And  $y$

$x \wedge x$

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

$x$  Or  $y$

$x \vee y$

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

Not( $x$ )

$\neg x$

$x$	Not
0	1
1	0

# Boolean Expressions

Not(0 Or (1 And 1)) =

Not(0 Or 1) =

Not(1) =

0

# 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	

$$\begin{aligned} (0 \text{ And } 0) \text{ Or } (\text{Not}(0) \text{ And } 1) &= \\ 0 \text{ Or } (1 \text{ And } 1) &= \\ 0 \text{ Or } 1 = 1 \end{aligned}$$

# 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

# Boolean Identities

- $(x \text{ And } y) = (y \text{ And } x)$
  - $(x \text{ Or } y) = (y \text{ Or } x)$
- } commutative laws
- $(x \text{ And } (y \text{ And } z)) = ((x \text{ And } y) \text{ And } z)$
  - $(x \text{ Or } (y \text{ Or } z)) = ((x \text{ Or } y) \text{ Or } z)$
- } associative laws
- $(x \text{ And } (y \text{ Or } z)) = (x \text{ And } y) \text{ Or } (x \text{ And } z)$
  - $(x \text{ Or } (y \text{ And } z)) = (x \text{ Or } y) \text{ And } (x \text{ Or } z)$
- } distributive laws
- $\text{Not}(x \text{ And } y) = \text{Not}(x) \text{ Or } \text{Not}(y)$
  - $\text{Not}(x \text{ Or } y) = \text{Not}(x) \text{ And } \text{Not}(y)$
- } De Morgan laws

# Boolean Algebra

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

De Morgan law

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

associative law

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

idempotence

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

De Morgan law

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

double negation

$$x \text{ Or } y$$

# Boolean Algebra

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



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



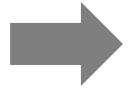
$x \text{ Or } y$

# Chapter 1: Boolean logic

---



Boolean logic



Boolean function synthesis

- Hardware description language
- Hardware simulation
- Multi-bit buses
- Project 1 overview

# Boolean expression → truth table

---

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



$x$	$y$	$z$	$f$
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

# Boolean expression ← truth table

---

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



$x$	$y$	$z$	$f$
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

# 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))

# From truth table to a Boolean expression

---

$x$	$y$	$z$	$f$
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))

# From truth table to a Boolean expression

---

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

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

# From truth table to a Boolean expression

---

$x$	$y$	$z$	$f$
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( $z$ ))

# 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

(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( $z$ )) =

# 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

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

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

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

# Theorem

---

Lemma: Any Boolean function can be represented using an expression containing And, Or And Not operations.

**Proof:**

Use the truth table to Boolean expression method

Lemma: Any Boolean function can be represented using an expression containing And and Not operations.

**Proof:**

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

**Can we do better than this?**

# Nand

---

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

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

## Theorem (revisited)

---

Lemma: Any Boolean function can be represented using an expression containing And, Or And Not operations.

**Proof:**

Use the truth table to Boolean expression method

Lemma: Any Boolean function can be represented using an expression containing And and Not operations.

**Proof:**

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

Theorem: Any Boolean function can be represented using an expression containing Nand operations only.

**Proof:**

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

# Chapter 1: Boolean logic

---

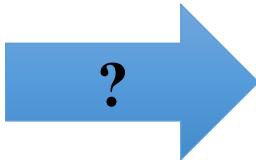
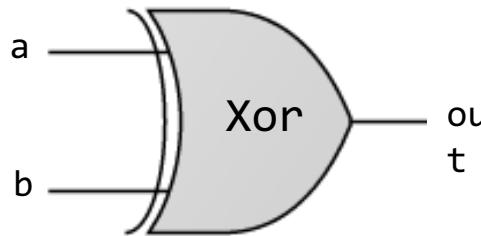
✓ Boolean logic

✓ Boolean function synthesis

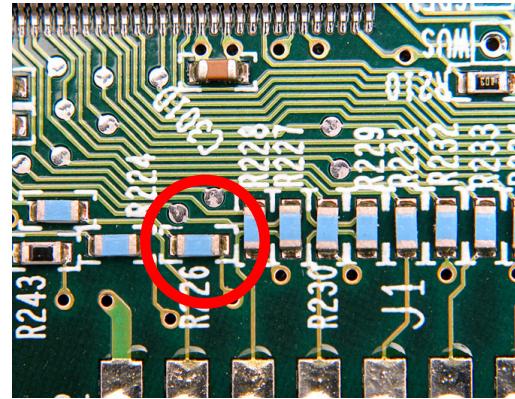
→ Hardware description language

- Hardware simulation
- Multi-bit buses
- Project 1 overview

# Building a logic gate



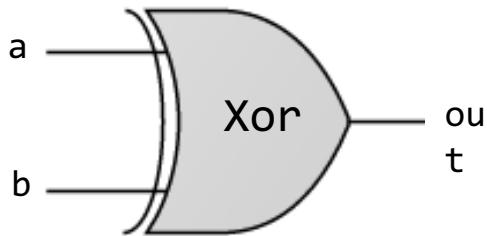
outputs 1 if one, and only one, of its inputs, is 1.



## 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.

# Design: from 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) */

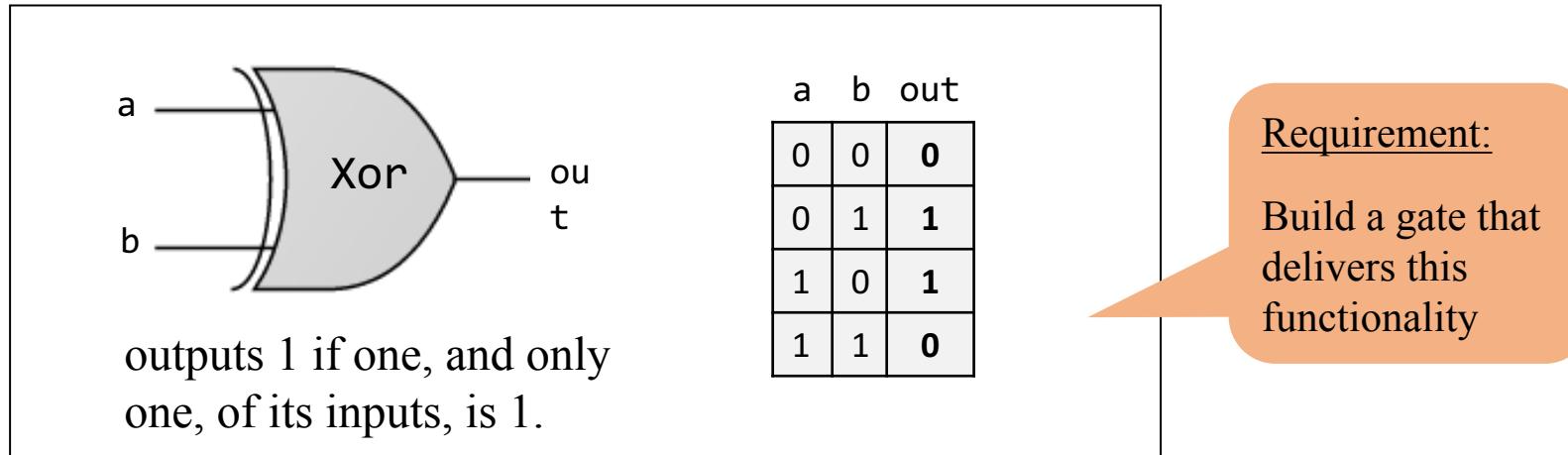
CHIP Xor {
    IN a, b;
    OUT out;

    PARTS:
        // Implementation missing
}
```

Gate interface

Expressed as an HDL stub file

# Design: from requirements to gate diagram



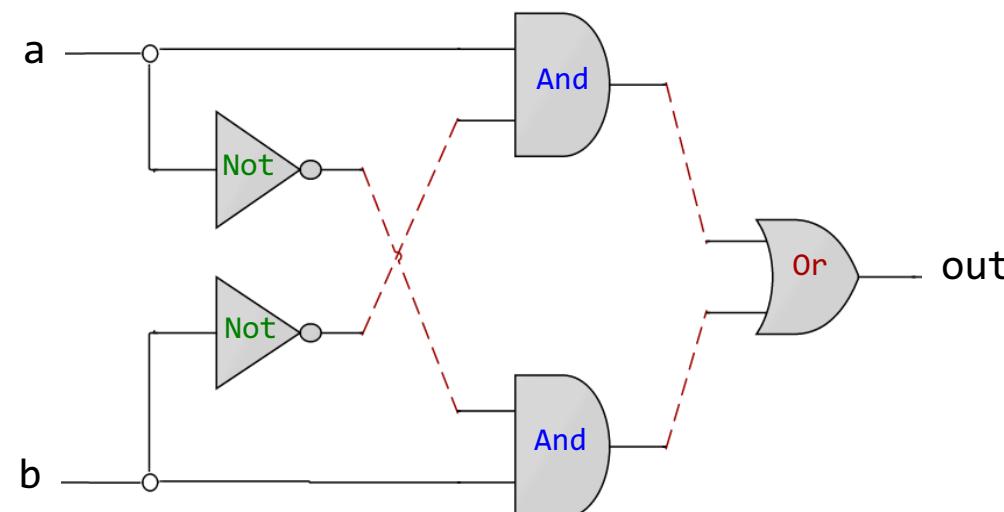
General idea:

$out=1$  when:

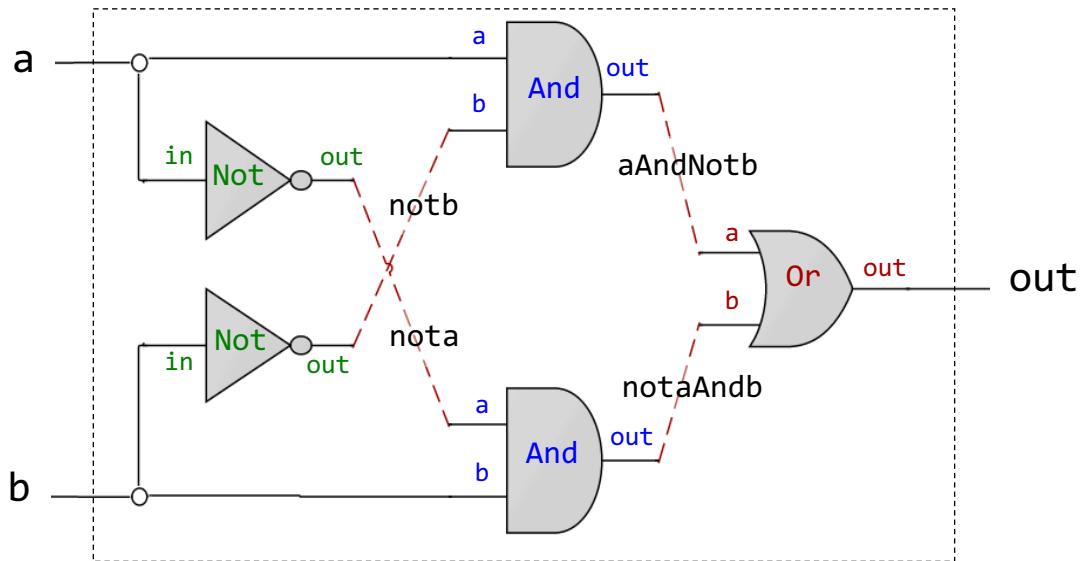
$a \text{ And } \text{Not}(b)$

Or

$b \text{ And } \text{Not}(a)$

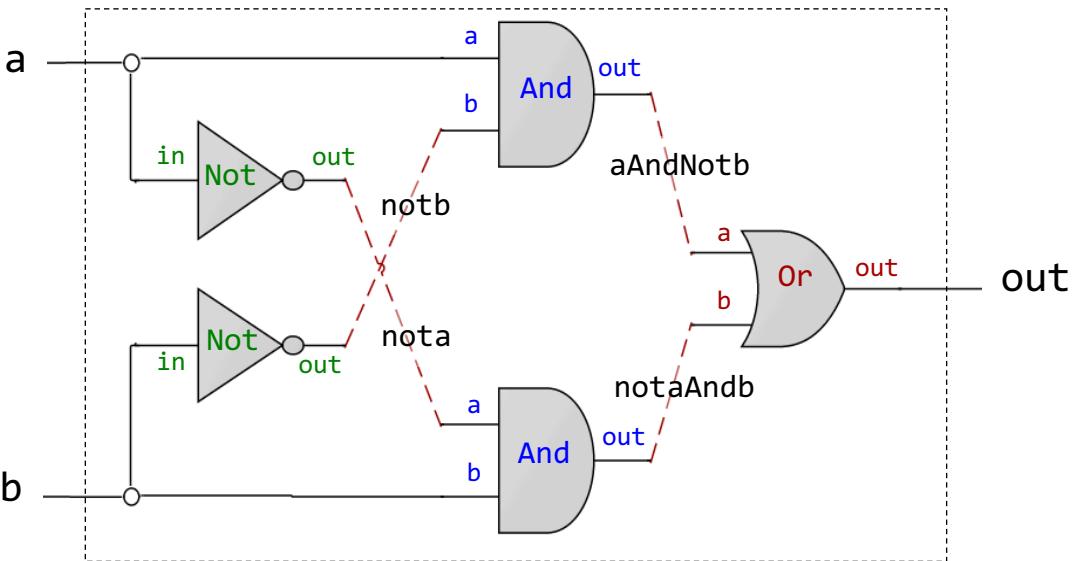


# Design: from gate diagram to HDL



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

# Design: from gate diagram to HDL



interface {  
    /\* 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);  
    }  
}

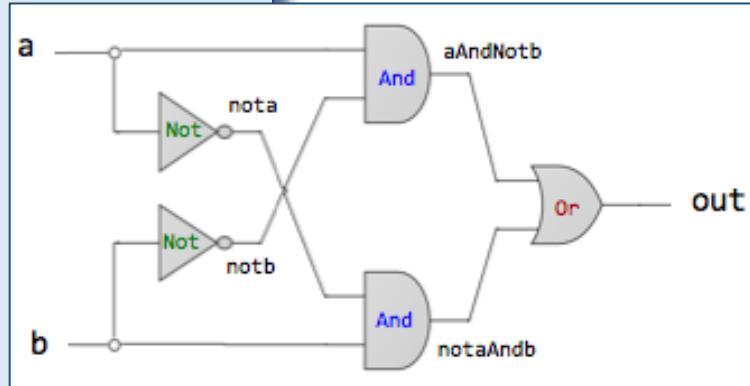
implementation {  
    Other Xor implementations  
    are possible!  
}

# HDL: some comments

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



- HDL is a functional / declarative language
- The order of HDL statements is insignificant
- Before using a chip part, you must know its interface. For example:  
`Not(in= ,out=), And(a= ,b= ,out= ), Or(a= ,b= ,out= )`
- Connection patterns like `chipName(a=a,...)` and `chipName(...,out=out)` are common

# Hardware description languages

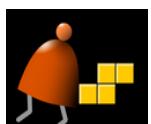
---

## Common HDLs:

- VHDL
- Verilog
- Many more HDLs...

## Our HDL

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



- Textbook / Appendix A
- [www.nand2tetris.org](http://www.nand2tetris.org) / HDL Survival Guide

# Chapter 1: Boolean logic

---

✓ Boolean logic

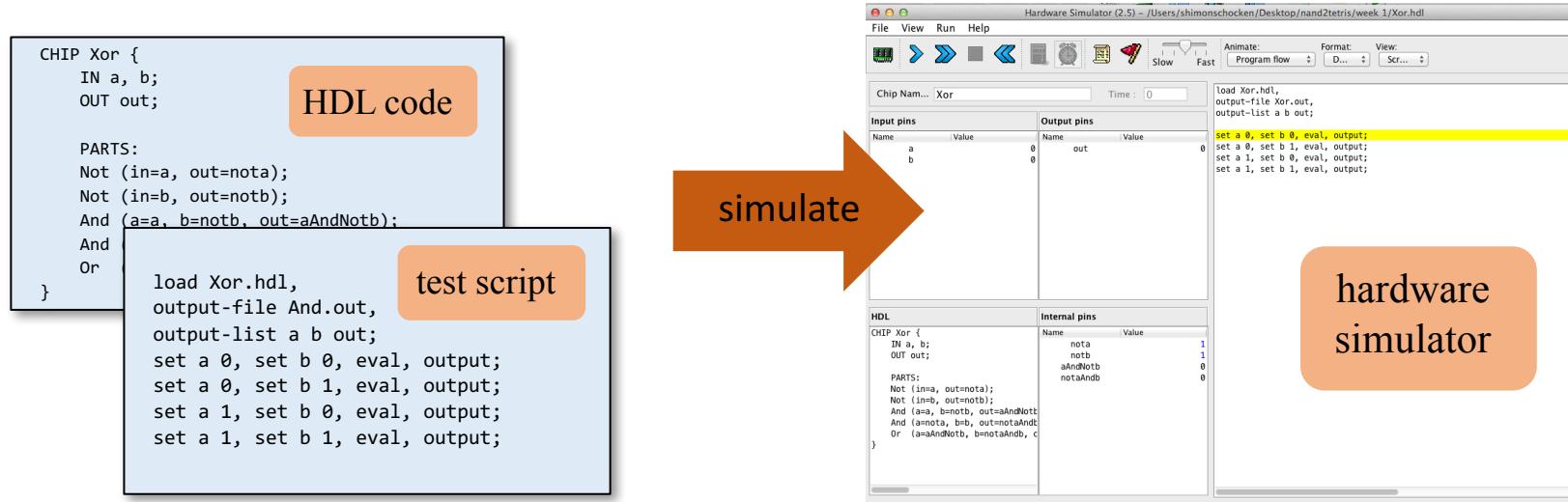
✓ Boolean function synthesis

✓ Hardware description language

→ Hardware simulation

- Multi-bit buses
- Project 1 overview

# Hardware simulation in a nutshell



## Simulation options:

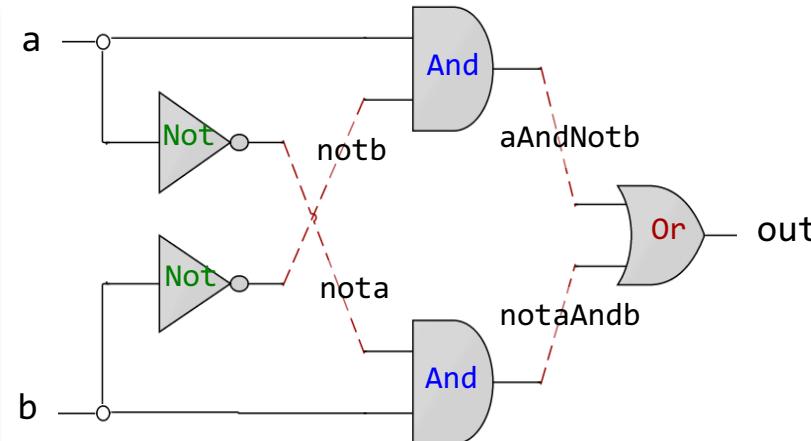
- Interactive
- Script-based
- With / without output and compare files

# Interactive simulation (using Xor as an example)

Xor.hdl

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

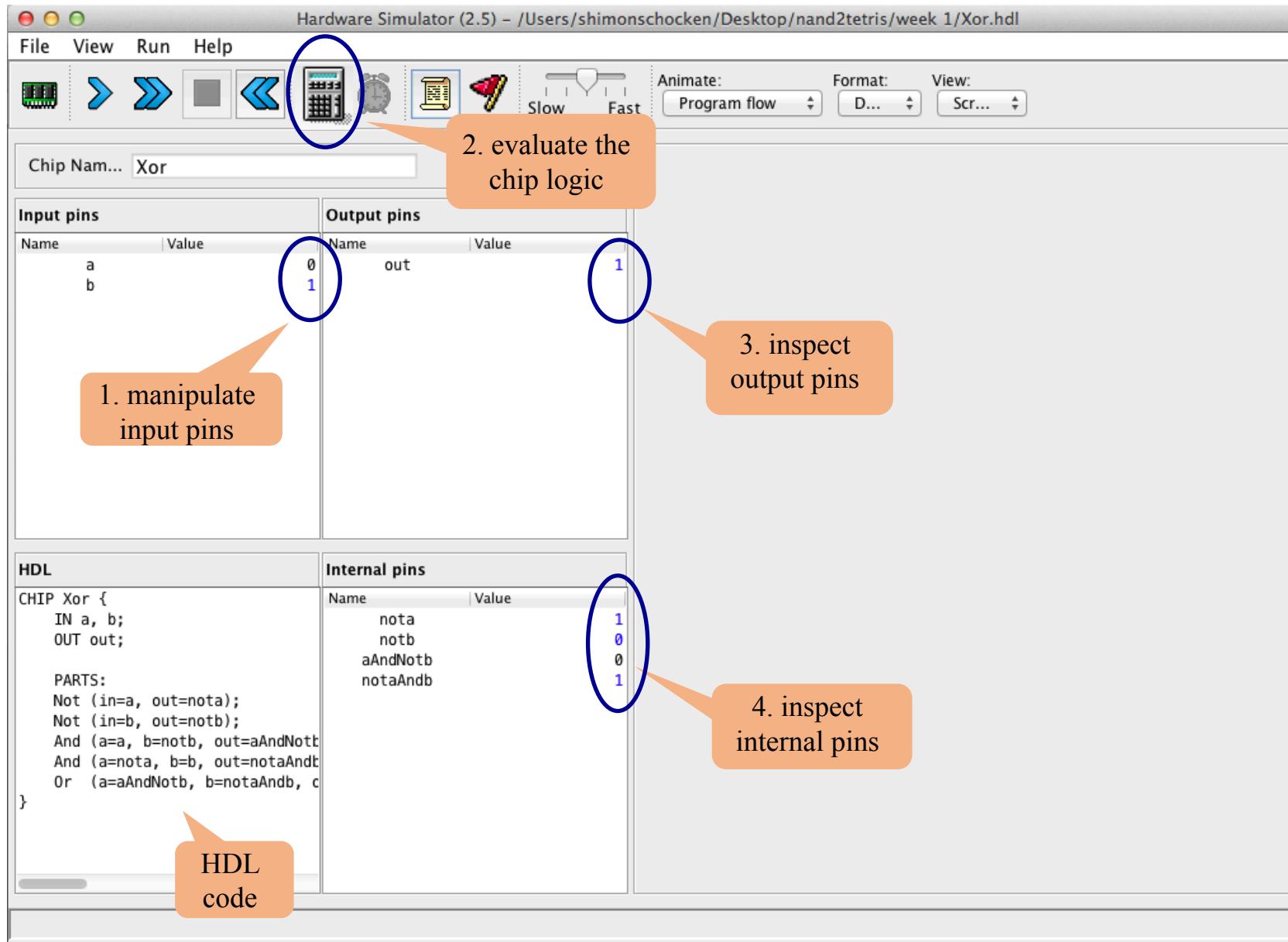
    PARTS:
        Not (in=a, out=nota);
        Not (in=b, out=notb);
        And (a=a, b=notb, out=aAndNotb);
        And (a=nota, b=b, out=notaAndb);
        Or (a=aAndNotb, b=notaAndb, out=out);
}
```



## Simulation process:

- Load the HDL file into the hardware simulator
- Enter values (0's and 1's) into the chip's input pins (e.g. *a* and *b*)
- Evaluate the chip's logic
- Inspect the resulting values of:
  - Output pins (e.g. *out*)
  - Internal pins (e.g. *nota*, *notb*, *aAndNotb*, *notaAndb*)

# Interactive simulation



# Interactive simulation

---



# Script-based simulation

Xor.hdl

```
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);  
}
```

tested  
chip

Xor.tst

```
load Xor.hdl;  
set a 0, set b 0, eval;  
set a 0, set b 1, eval;  
set a 1, set b 0, eval;  
set a 1, set b 1, eval;
```

test script = series of  
commands to the simulator

## Benefits:

- “Automatic” testing
- Replicable testing

# Script-based simulation, with an output file

Xor.hdl

```
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);  
}
```

tested  
chip

Xor.tst

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

test  
script

## The logic of a typical test script

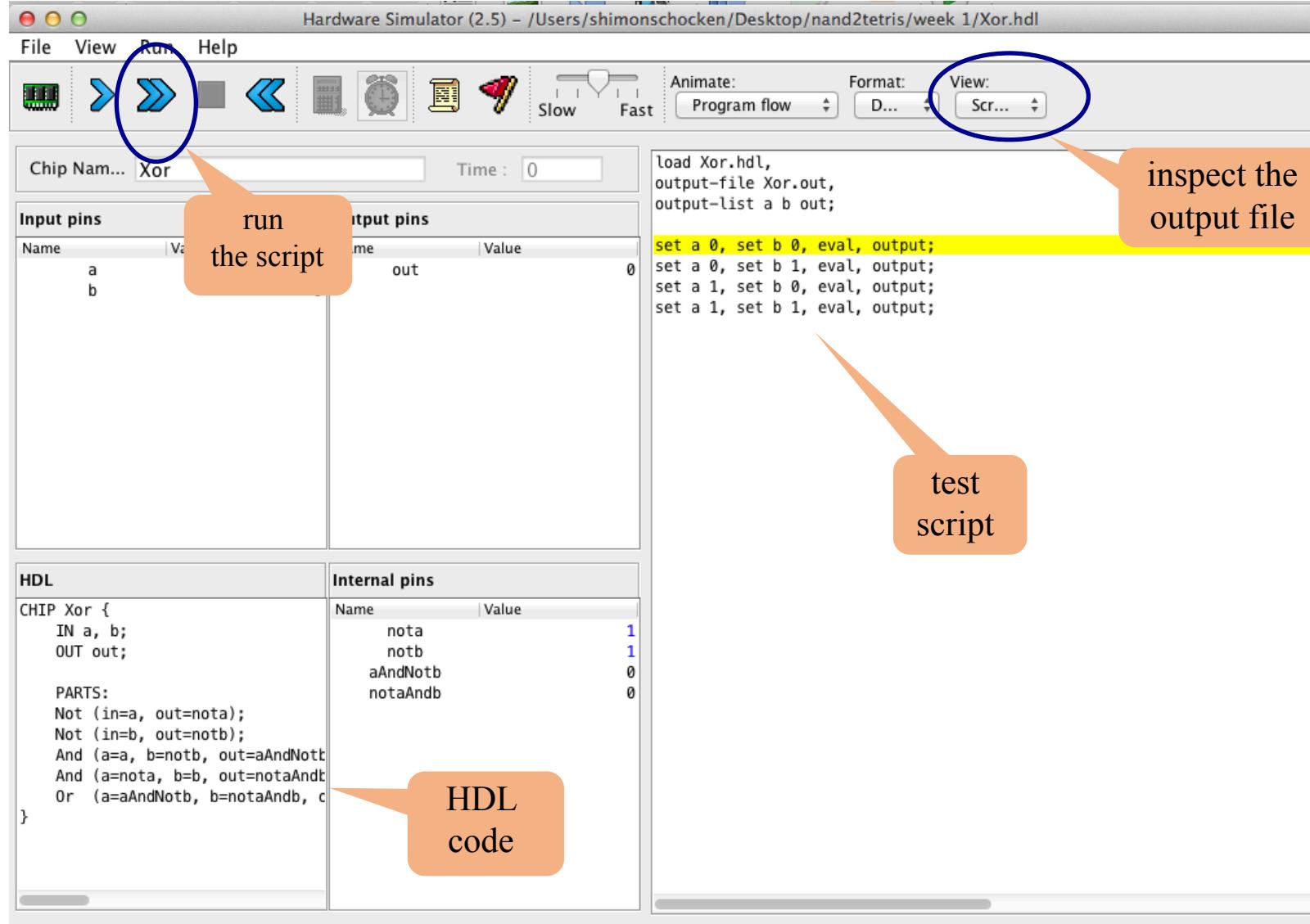
- Initialize:
  - Load an HDL file
  - Create an empty output file
  - List the names of the pins whose values will be written to the output file
- Repeat:
  - **set – eval - output**

Xor.out

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

Output File, created by the test script as a side-effect of the simulation process

# Script-based simulation



# Script-based simulation

---



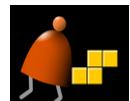
# Hardware simulators

---

- There are many of them!

## Our hardware simulator

- Minimal and simple
- Provides all you need for this course
- Hardware simulator documentation:



[www.nand2tetris.org](http://www.nand2tetris.org) / Hardware Simulator Tutorial

# Revisiting script-based simulation with output files

Xor.hdl

```
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);  
}
```

tested  
chip

Xor.tst

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

test  
script

Output file, created by the  
test script as a side-effect of  
the simulation process

Xor.out

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

# Script-based simulation, with compare files

Xor.hdl

```
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);  
}
```

tested chip

Xor.tst

```
load Xor.hdl,  
output-file Xor.out,  
compare-to Xor.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;
```

test script

## Simulation-with-compare-file logic

- When each output command is executed, the outputted line is compared to the corresponding line in the compare file
- If the two lines are not the same, the simulator throws a comparison error.

Xor.cmp

Xor.out		a	b	out
0	0	0	0	0
0	1	0	1	1
1	0	1	0	1
1	1	1	1	0

# Behavioral simulation

Xor.hdl

```
CHIP Xor {  
    IN a, b;  
    OUT out;  
  
    BUILTIN Xor  
        // Built-in chip implementation,  
        // can execute in the hardware  
        // simulator like any other chip.  
}
```

built-in chip implementation

Xor.tst

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

test script

## Behavioral simulation:

- The chip logic (abstraction) can be implemented in some high-level language
- Enables high-level planning and testing of a hardware architecture before writing any HDL code.

Xor.out

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

Xor.cmp

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

# Hardware construction projects

---

- The players (first approximation):
  - System architects
  - Developers
- The system architect decides which chips are needed
- For each chip, the architect creates
  - A chip API
  - A test script
  - A compare file
- Given these resources, the developers can build the chips.

# The developer's view (of, say, a xor gate)

Xor.hdl

```
/** returns 1 if (a != b) */  
  
CHIP Xor {  
    IN a, b;  
    OUT out;  
  
    PARTS:  
        // Implementation missing  
}
```

stub  
file

Xor.tst

```
load Xor.hdl,  
output-file Xor.out,  
compare-to Xor.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;
```

test  
script

- Taken together, the three files provide a convenient specification of:
  - The chip interface (.hdl)
  - What the chip is supposed to do (.cmp)
  - How to test the chip (.tst)
- The developer's task:  
implement the chip, using these resources.

Xor.cmp

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

compare  
file

# Chapter 1: Boolean logic

---

- ✓ Boolean logic
- ✓ Boolean function synthesis
- ✓ Hardware description language
- ✓ Hardware simulation
- Multi-bit buses
  - Project 1 overview

# Arrays of Bits

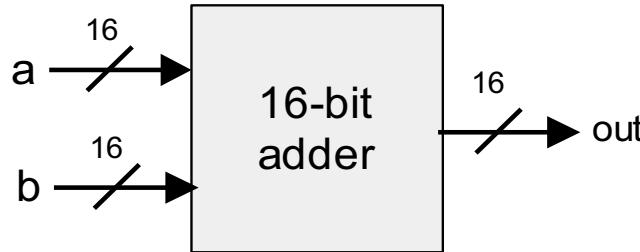
---

- 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"
- HDLs usually provide notation and means for handling buses.

# Example: adding 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 individual bits within buses

---

```
/*
 * 4-way And: Ands 4 bits.
 */
CHIP And4Way {
    IN 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);
}
```

# Working with individual bits within buses

---

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

# 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 our HDL

- buses are indexed right to left: if `foo` is a 16-bit bus, then `foo[0]` is the right-most bit, and `foo[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.

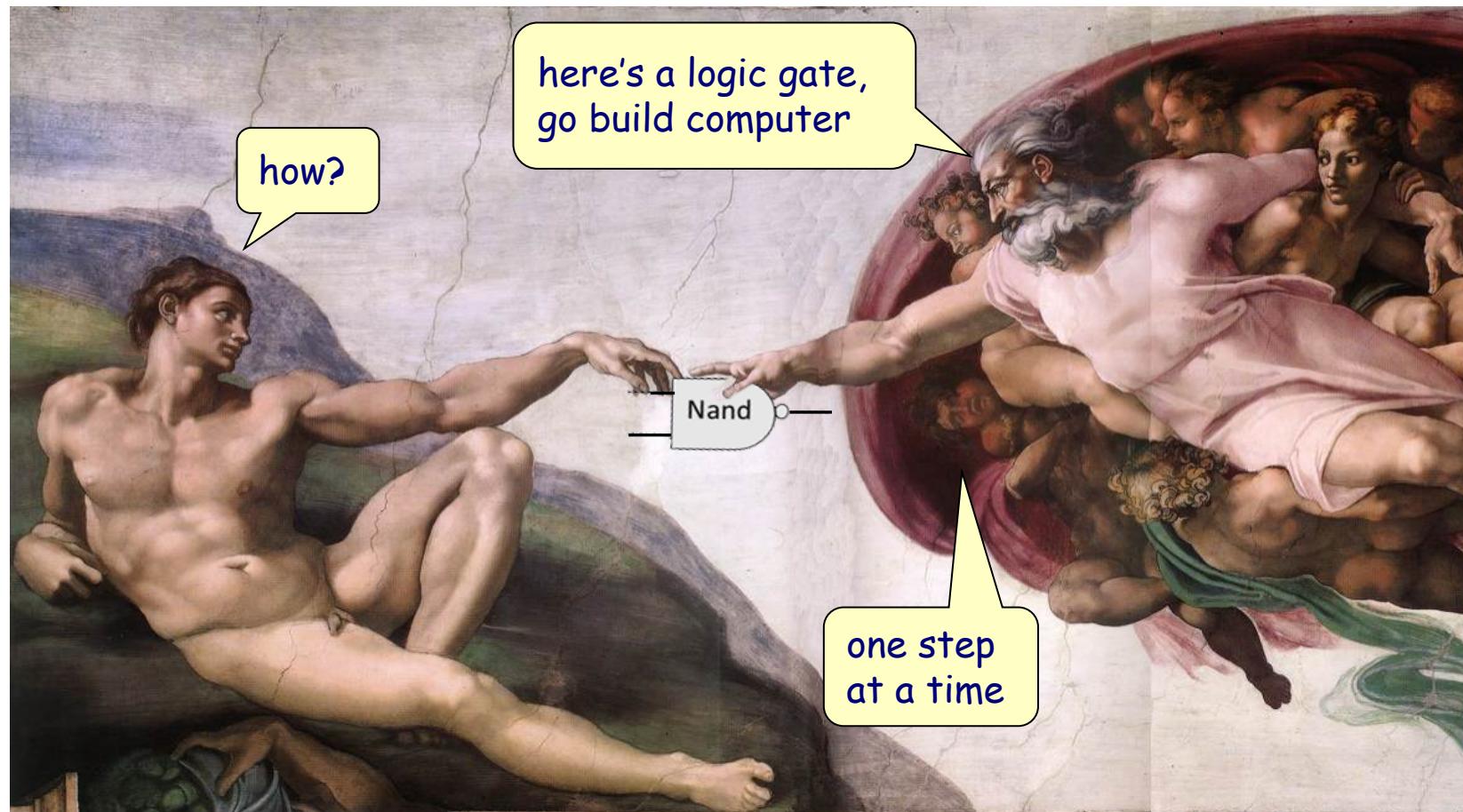
# Chapter 1: Boolean logic

---

- ✓ Boolean logic
  - ✓ Boolean function synthesis
  - ✓ Hardware description language
  - ✓ Hardware simulation
  - ✓ Multi-bit buses
- Project 1 overview

# Nand to Tetris course methodology

---



## Project 1: the first step

# Project 1

Given: Nand

Goal: Build the following gates:

<u>Elementary logic gates</u>	<u>16-bit variants</u>	<u>Multi-way variants</u>
<input type="checkbox"/> Not	<input type="checkbox"/> Not16	<input type="checkbox"/> Or8Way
<input type="checkbox"/> And	<input type="checkbox"/> And16 ←	<input type="checkbox"/> Mux4Way16 ←
<input type="checkbox"/> Or	<input type="checkbox"/> Or16	<input type="checkbox"/> Mux8Way16
<input type="checkbox"/> Xor	<input type="checkbox"/> Mux16	<input type="checkbox"/> DMux4Way
<input type="checkbox"/> Mux ←		<input type="checkbox"/> DMux8Way
<input type="checkbox"/> DMux ←		

Why these 15 particular gates?

- Commonly used gates
- Comprise all the elementary logic gates needed to build our computer.

# Project 1

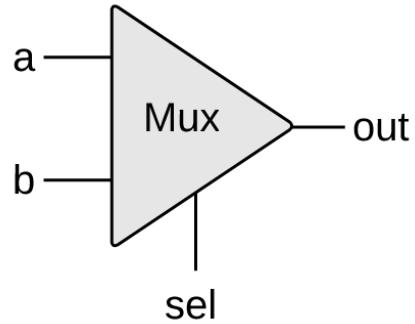
---

Given: Nand

Goal: Build the following gates:

<u>Elementary logic gates</u>	<u>16-bit variants</u>	<u>Multi-way variants</u>
<input type="checkbox"/> Not	<input type="checkbox"/> Not16	<input type="checkbox"/> Or8Way
<input type="checkbox"/> And	<input type="checkbox"/> And16	<input type="checkbox"/> Mux4Way16
<input type="checkbox"/> Or	<input type="checkbox"/> Or16	<input type="checkbox"/> Mux8Way16
<input type="checkbox"/> Xor	<input type="checkbox"/> Mux16	<input type="checkbox"/> DMux4Way
<input type="checkbox"/> Mux		<input type="checkbox"/> DMux8Way
<input type="checkbox"/> DMux		

# Multiplexor



```
if (sel==0)
    out=a
else
    out=b
```

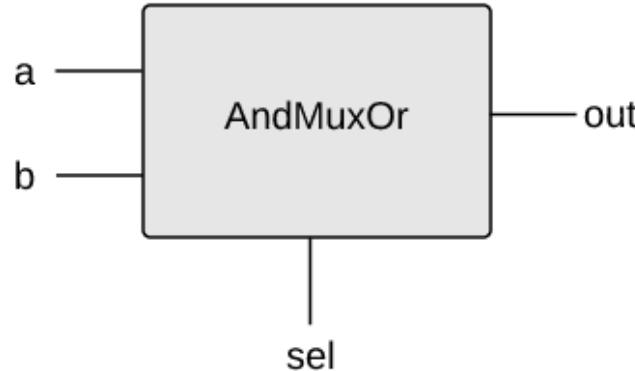
a	b	sel	out
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	1
0	0	1	0
0	1	1	1
1	0	1	0
1	1	1	1

sel	out
0	a
1	b

abbreviated  
truth table

- A 2-way multiplexor enables selecting, and outputting, one of two possible inputs
- Widely used in:
  - Digital design
  - Communications networks

# Example: using mux logic to build a programmable gate

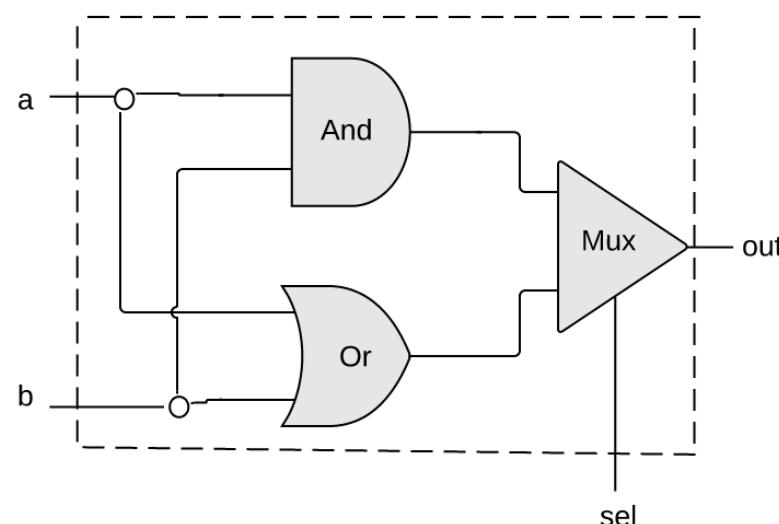


```
if (sel==0)
    out = (a And b)
else
    out = (a Or b)
```

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

When  $\text{sel}==0$   
the gate acts like  
an And gate

When  $\text{sel}==1$   
the gate acts like  
an Or gate



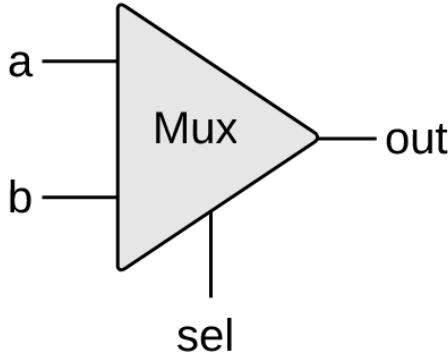
Mux.hdl

```
CHIP AndMuxOr {
    IN a, b, sel;
    OUT out;

    PARTS:
        And (a=a, b=b, out=andOut);
        Or (a=a, b=b, out=orOut);
        Mux (a=andOut, b=orOut, sel=sel, out=out);
}
```

# Multiplexor implementation

---



```
if (sel==0)
    out=a
else
    out=b
```

sel	out
0	a
1	b

Mux.hdl

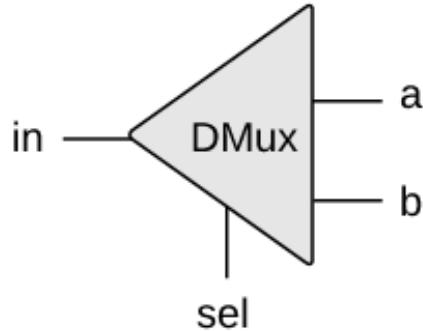
```
CHIP Mux {
    IN a, b, sel;
    OUT out;

    PARTS:
        // Put your code here:
}
```

Implementation tip:  
Can be implemented with  
And, Or, and Not gates

# Demultiplexor

---



```
if (sel==0)
    {a,b}={in,0}
else
    {a,b}={0,in}
```

in	sel	a	b
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

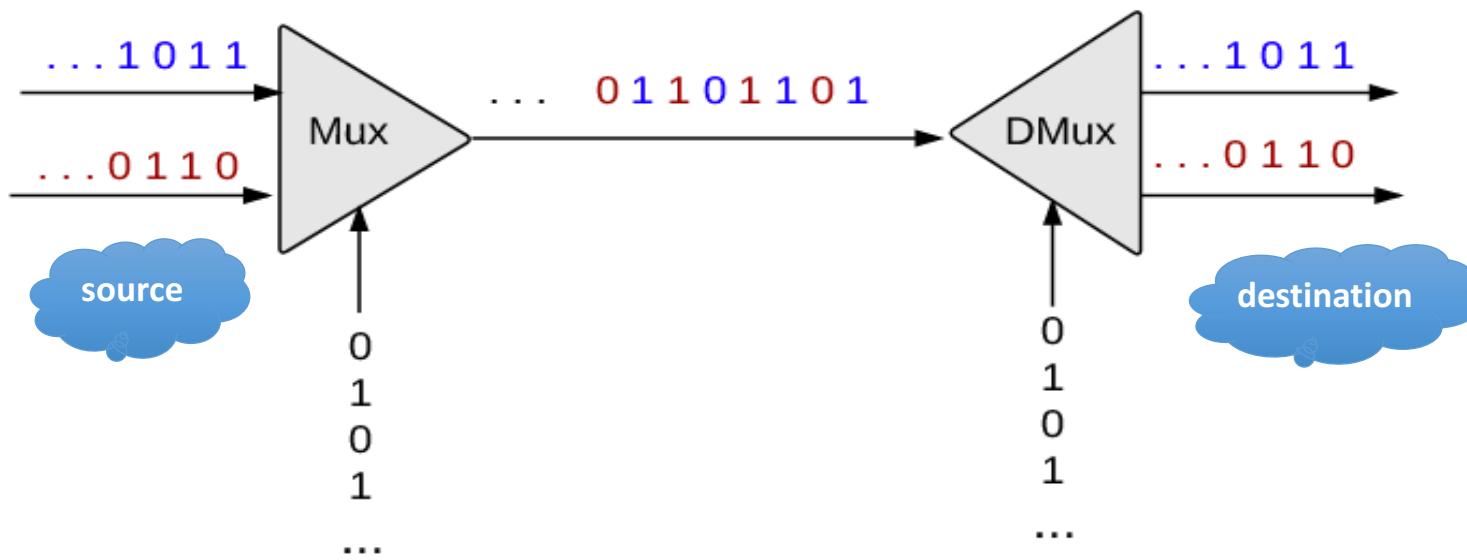
- Acts like the “inverse” of a multiplexor
- Distributes the single input value into one of two possible destinations

DMux.hdl

```
CHIP DMux {
    IN in, sel;
    OUT a, b;

    PARTS:
        // Put your code here:
}
```

## Example: Multiplexing / demultiplexing in communications networks



- Each `sel` bit is connected to an oscillator that produces a repetitive train of alternating 0 and 1 signals
- Enables transmitting multiple messages on a single, shared communications line
- A common use of multiplexing / demultiplexing logic
- Unrelated to this course.

# Project 1

---

## Elementary logic gates

- Not
- And
- Or
- Xor
- Mux
- DMux

## 16-bit variants

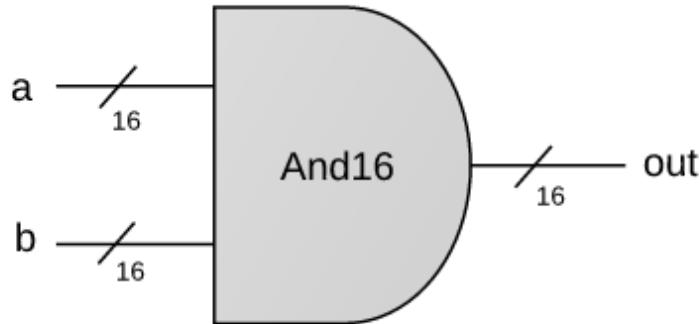
- Not16
- And16 
- Or16
- Mux16

## Multi-way variants

- Or8Way
- Mux4Way16
- Mux8Way16
- DMux4Way
- DMux8Way

# And16

---



```
CHIP And16 {  
    IN a[16], b[16];  
    OUT out[16];  
  
    PARTS:  
        // Put your code here:  
}
```

$$\begin{array}{r} a = 1\ 0\ 1\ 0\ 1\ 0\ 1\ 1\ 0\ 1\ 0\ 1\ 1\ 1\ 0\ 0 \\ b = 0\ 0\ 1\ 0\ 1\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 1\ 0\ 1\ 0 \\ \hline \text{out} = 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0 \end{array}$$

- A straightforward 16-bit extension of the elementary And gate  
(See previous slides on working with multi-bit buses)

# Project 1

---

## Elementary logic gates

- Not
- And
- Or
- Xor
- Mux
- DMux

## 16-bit variants

- Not16
- And16
- Or16
- Mux16

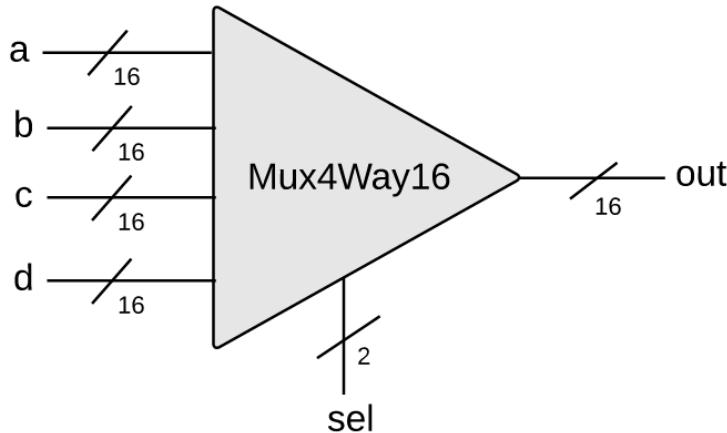
## Multi-way variants

- Or8Way
- Mux4Way16
- Mux8Way16
- DMux4Way
- DMux8Way



# 16-bit, 4-way multiplexor

---



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

Mux4Way16.hdl

```
CHIP Mux4Way16 {
    IN a[16], b[16], c[16], d[16],
    sel[2];
    OUT out[16];

    PARTS:
        // Put your code here:
}
```

Implementation tip:

Can be built from several Mux16 gates

# Project 1

---

Given: Nand

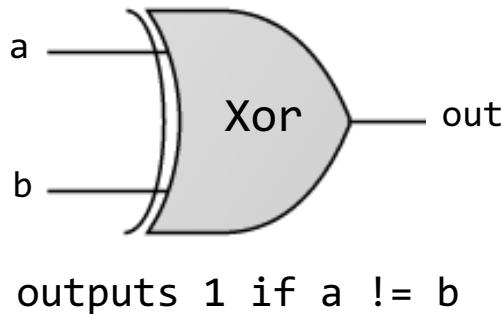
Goal: Build the following gates:

<u>Elementary logic gates</u>	<u>16-bit variants</u>	<u>Multi-way variants</u>
<input type="checkbox"/> Not	<input type="checkbox"/> Not16	<input type="checkbox"/> Or8Way
<input type="checkbox"/> And	<input type="checkbox"/> And16	<input type="checkbox"/> Mux4Way16
<input type="checkbox"/> Or	<input type="checkbox"/> Or16	<input type="checkbox"/> Mux8Way16
<input type="checkbox"/> Xor	<input type="checkbox"/> Mux16	<input type="checkbox"/> DMux4Way
<input type="checkbox"/> Mux		<input type="checkbox"/> DMux8Way
<input type="checkbox"/> DMux		



So how to actually build  
these gates?

# Chip building materials (using xor as an example)



Xor.cmp

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

The contract:

When running your Xor.hdl on  
the supplied Xor.tst,  
your Xor.out should be the  
same as the supplied Xor.cmp

Xor.hdl

```
CHIP Xor {  
    IN a, b;  
    OUT out;  
  
    PARTS:  
        // Put your code here.  
}
```

Xor.tst

```
load Xor.hdl,  
output-file Xor.out,  
compare-to Xor.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;
```

# Project 1 Resources

**From NAND to Tetris**  
*Building a Modern Computer From First Principles*  
[www.nand2tetris.org](http://www.nand2tetris.org)



Home  
Prerequisites  
Syllabus  
**Course**  
Book  
Software  
Terms  
Papers  
Talks  
Cool Stuff  
About  
Team  
Q&A

## Project 1: Elementary Logic Gates

### Background

A typical computer architecture is based on a set of elementary logic gates like And, Or, Mux, etc., as well as their bit-wise versions And16, Or16, Mux16, etc. (assuming a 16-bit machine). This project engages you in the construction of a typical set of basic logic 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)	Description	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
And16	16-bit And	And16.tst	And16.cmp
Or16	16-bit Or	Or16.tst	Or16.cmp
Mux16	16-bit multiplexor	Mux16.tst	Mux16.cmp
DMux16	16-bit DMux	DMux16.tst	DMux16.cmp

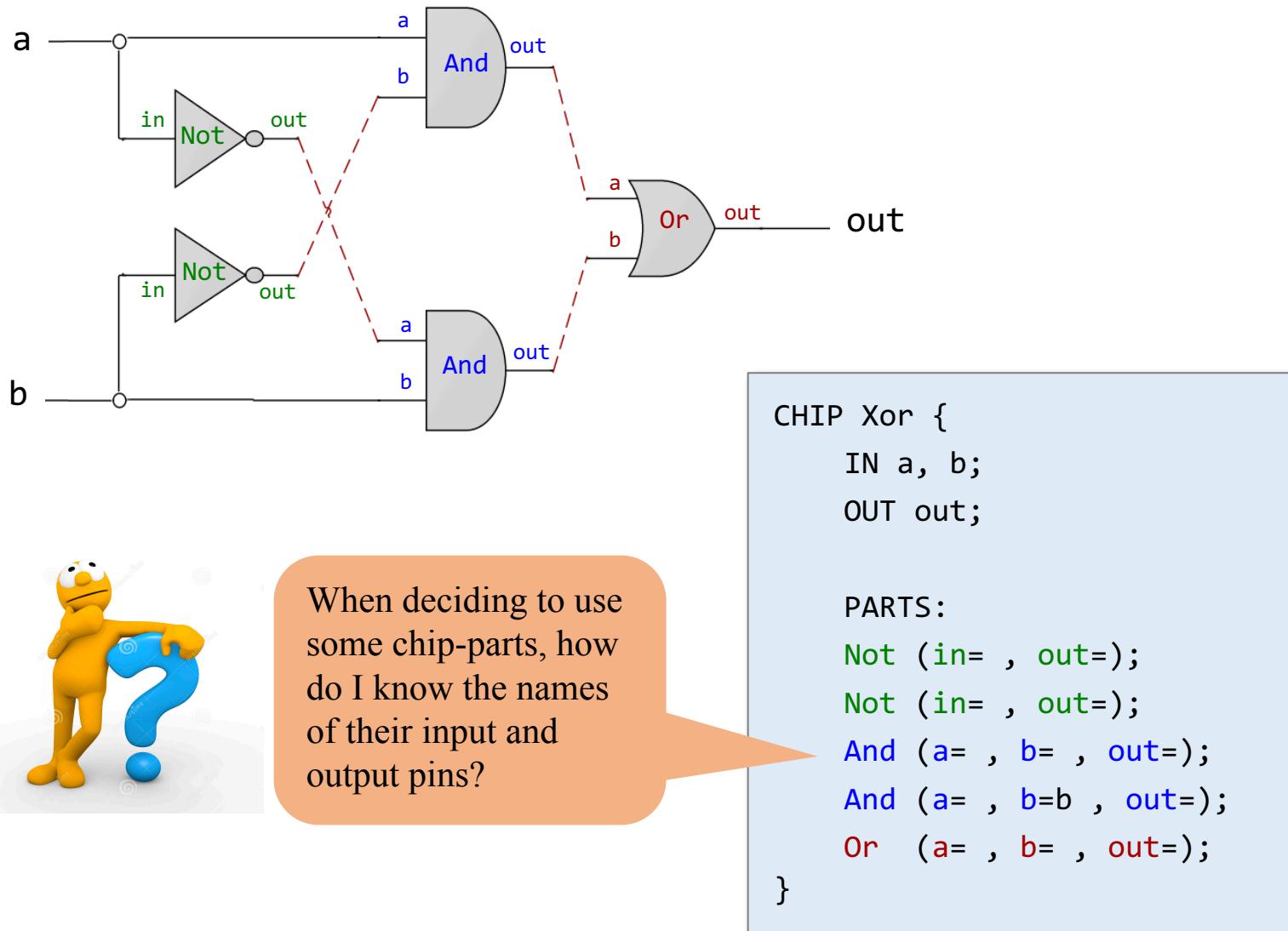
All the necessary project 1 files  
are available in:  
**nand2tetris/projects/01**

## More resources

---

- Text editor (for writing your HDL files)
  - HDL Survival Guide
  - Hardware Simulator Tutorial
  - nand2tetris Q&A forum
- All available in: [www.nand2tetris.org](http://www.nand2tetris.org)

# Hack chipset API



# Hack chipset API

```
Add16 (a= ,b= ,out= );
ALU (x= ,y= ,zx= ,nx= ,zy= ,ny= ,f= ,no= ,out= ,zr= ,ng= );
And16 (a= ,b= ,out= );
And (a= ,b= ,out= );
Aregister (in= ,load= ,out= );
Bit (in= ,load= ,out= );
CPU (inM= ,instruction= ,reset= ,outM= );
DFF (in= ,out= );
DMux4Way (in= ,sel= ,a= ,b= ,c= ,d= );
DMux8Way (in= ,sel= ,a= ,b= ,c= ,d= );
Dmux (in= ,sel= ,a= ,b= );
Dregister (in= ,load= ,out= );
FullAdder (a= ,b= ,c= ,sum= ,carry= );
HalfAdder (a= ,b= ,sum= , carry= );
Inc16 (in= ,out= );
Keyboard (out= );
Memory (in= ,load= ,address= ,out= );
Mux16 (a= ,b= ,sel= ,out= );
Mux4Way16 (a= ,b= ,c= ,d= ,sel= ,out= );
Mux8Way16 (a= ,b= ,c= ,d= ,e= ,f= ,g= );
Mux8Way (a= ,b= ,c= ,d= ,e= ,f= ,g= ,h= ,sel= ,out= );
Mux (a= ,b= ,sel= ,out= );
Nand (a= ,b= ,out= );
Not16 (in= ,out= );
Not (in= ,out= );
Or16 (a= ,b= ,out= );
Or8Way (in= ,out= );
Or (a= ,b= ,out= );
PC (in= ,load= ,inc= ,reset= ,out= );
PCLoadLogic (cinstr= ,j1= ,j2= ,j3= ,load= ,inc= );
RAM16K (in= ,load= ,address= ,out= );
RAM4K (in= ,load= ,address= ,out= );
RAM512 (in= ,load= ,address= ,out= );
RAM64 (in= ,load= ,address= ,out= );
RAM8 (in= ,load= ,address= ,out= );
Register (in= ,load= ,out= );
ROM32K (address= ,out= );
Screen (in= ,load= ,address= ,out= );
Xor (a= ,b= ,out= );
```



(see *HDL Survival Guide* @ [www.nand2tetris.org](http://www.nand2tetris.org))

# Built-in chips

---

```
CHIP Foo {  
    IN ...;  
    OUT ...;  
  
    PARTS:  
    ...  
    Mux16(...)  
    ...  
}
```

Q: What happens if there is no `Mux16.hdl` file in the current directory?

A: The simulator invokes, and evaluates, the built-in version of `Mux16` (if such exists).

- The supplied simulator software features built-in chip implementations of all the chips in the Hack chip set
- If you don't implement some chips from the Hack chipset, you can still use them as chip-parts of other chips:
  - Just rename their given stub files to, say, `Mux16.hdl1`
  - This will cause the simulator to use the built-in chip implementation.

# Best practice advice

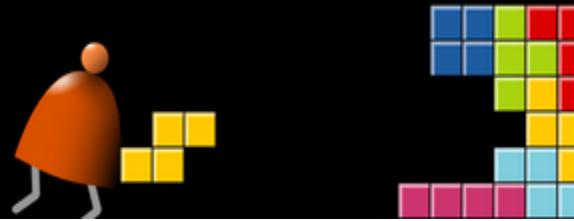
---

- Try to implement the chips in the given order
- If you don't implement some chips, you can still use them as chip-parts in other chips (the built-in implementations will kick in)
- You can invent new, “helper chips”; however, this is not required: you can build any chip using previously-built chips only
- Strive to use as few chip-parts as possible.

# Chapter 1: Boolean logic

---

- ✓ Boolean logic
- ✓ Boolean function synthesis
- ✓ Hardware description language
- ✓ Hardware simulation
- ✓ Multi-bit buses
- ✓ Project 1 overview



## Chapter 1

# Boolean Logic

These slides support chapter 1 of the book

*The Elements of Computing Systems*

By Noam Nisan and Shimon Schocken

MIT Press