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, 2021

Chapter 1: Boolean logic

Theory

- Basic concepts
- Boolean algebra
- Boolean functions
- Nand

Practice

- Logic gates
- HDL
- Hardware simulation
- Multi-bit buses

Project 1

- Introduction
- Chips
- Guidelines

Boolean values

	`off`	`no`	`false`	`0`
	`on`	`yes`	`true`	`1`

<u>George Boole</u>

1815 - 1864

Different labels, all referring to two Boolean / binary values.

Boolean values

$\ldots \; b_2 \quad b_1 \quad b_0$

1 binary variable: 2 possible states

2 binary variables: 4 possible states

3 binary variables: 8 possible states

. . .

<u>Question</u>: How many different states can be represented by *N* binary variables?

Boolean values

… b_2 b_1 b_0

1 binary variable: 2 possible states

2 binary variables: 4 possible states

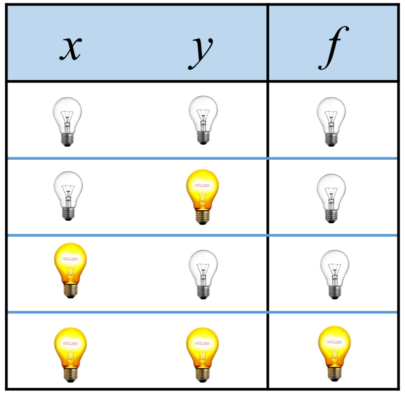
3 binary variables: 8 possible states

…

<u>Question</u>: How many different states can be represented by N binary variables?

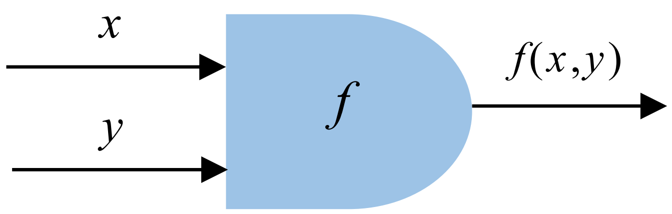
<u>Answer</u>: 2^N

Boolean functions



Boolean functions

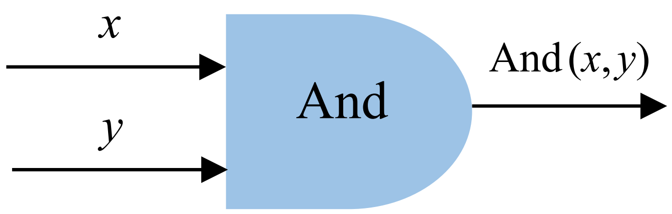
x	y	f
0	0	0
0	1	0
1	0	0
1	1	1



$$f(x,y) = \begin{cases} 1 & \text{when } x == 1 \text{ and } y == 1 \\ 0 & \text{otherwise} \end{cases}$$

Boolean functions

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



$$\text{And}(x,y) = \begin{cases} 1 & \text{when } x == 1 \text{ and } y == 1 \\ 0 & \text{otherwise} \end{cases}$$

<u>Boolean function</u> (like And(x,y)):

A function that operates on boolean variables, and returns a boolean value.

<u>Boolean operator</u> (like x And y):

A simple boolean function that operates on a few boolean variables, called *operands*.

Boolean functions

x And y

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

x Or y

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

Not(x)

x	Not
0	1
1	0

x Nand y

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

x Xor y

x	y	Xor
0	0	0
0	1	1
1	0	1
1	1	0

• • •

$x \boldsymbol{f} y$

x	y	f
0	0	v_1
0	1	v_2
1	0	v_3
1	1	v_4

Boolean functions

x And y

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

x Or y

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

Question:

How many Boolean functions $x \, \boldsymbol{f} \, y$ exist over two binary (2-valued) variables?

Answer: 16

N binary variables span 2^{2^N} Boolean functions.

x Nand y

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

x Xor y

x	y	Xor
0	0	0
0	1	1
1	0	1
1	1	0

. . .

$x \, \boldsymbol{f} \, y$

x	y	f
0	0	v_1
0	1	v_2
1	0	v_3
1	1	v_4

Boolean functions

x And y

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

x Or y

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

Not(x)

y	Not
0	1
1	0

Boolean function evaluation (example):

Not(x Or (y And z))

Evaluate this function for, say,
x = 0, y = 1, z = 1

Not(0 Or (1 And 1)) =

Not(0 Or 1) =

Not(1) =

0

Chapter 1: Boolean logic

<u>Theory</u>

- Basic concepts
- Boolean algebra
- Boolean functions
- Nand

<u>Practice</u>

- Logic gates
- HDL
- Hardware simulation
- Multi-bit buses

<u>Project 1</u>

- Introduction
- Chips
- Guidelines

Some Boolean identities

Commutative: x And y = y And x
x Or y = y Or x

Associative: x And (y And z) = (x And y) And z
x Or (y Or z) = (x Or y) Or z

Distributive: x And (y Or z) = (x And y) Or (x And z)
x Or (y And z)) = (x Or y) And (x Or z)

De Morgan: Not(x And y) = Not(x) Or Not(y)
Not(x Or y) = Not(x) And Not(y)

Idempotent: x And x = x
x Or x = x

Double negation: Not(Not(x)) = x

All these identities can be easily proved from the function definitions of And, Or, Not

For example, let's prove this identity

Boolean algebra

Prove

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

$f(x,y)$ = Not(x And y)

x	y	f
0	0	1
0	1	1
1	0	1
1	1	0

$g(x,y)$ = Not(x) Or Not(y)

x	y	g
0	0	1
0	1	1
1	0	1
1	1	0

Proof: Fill in the right column in both truth tables.

If $f = g$, the identity is proved.

Boolean algebra

Commutative: x And y = y And x
x Or y = y Or x

Associative: x And (y And z) = (x And y) And z
x Or (y Or z) = (x Or y) Or z

Distributive: x And (y Or z) = (x And y) Or (x And z)
x Or (y And z)) = (x Or y) And (x Or z)

De Morgan: Not(x And y) = Not(x) Or Not(y)
Not(x Or y) = Not(x) And Not(y)

Idempotent: x And x = x
x Or x = x

Double negation: Not(Not(x)) = x

Substitution:

In any such identity, x and y can be substituted with any boolean function

Substitution examples:

Not(Not(a)) = a

Not(Not(u Or v)) = u Or v

Etc.

Boolean algebra

Commutative: x And y = y And x
x Or y = y Or x

Associative: x And (y And z) = (x And y) And z
x Or (y Or z) = (x Or y) Or z

Distributive: x And (y Or z) = (x And y) Or (x And z)
x Or (y And z)) = (x Or y) And (x Or z)

De Morgan: Not(x And y) = Not(x) Or Not(y)
Not(x Or y) = Not(x) And Not(y)

Idempotent: x And x = x
x Or x = x

Double negation: Not(Not(x)) = x

Boolean algebra

Commutative: x And y = y And x

x Or y = y Or x

Associative: x And (y And z) = (x And y) And z

x Or (y Or z) = (x Or y) Or z

Distributive: x And (y Or z) = (x And y) Or (x And z)

x Or (y And z)) = (x Or y) And (x Or z)

De Morgan: Not(x And y) = Not(x) Or Not(y)

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

Idempotent: x And x = x

x Or x = x

Double negation: Not(Not(x)) = x

Task: Simplify this function (example):

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

Boolean algebra

Commutative: *x* And *y* = *y* And *x*
x Or *y* = *y* Or *x*

Associative: $\boxed{x \text{ And } (y \text{ And } z) =}$ (*x* And *y*) And *z*
x Or (*y* Or *z*) = (*x* Or *y*) Or *z*

Distributive: *x* And (*y* Or *z*) = (*x* And *y*) Or (*x* And *z*)
x Or (*y* And *z*)) = (*x* Or *y*) And (*x* Or *z*)

De Morgan: Not(*x* And *y*) = Not(*x*) Or Not(*y*)
Not(*x* Or *y*) = Not(*x*) And Not(*y*)

Idempotent: *x* And *x* = *x*
x Or *x* = *x*

Double negation: Not(Not(*x*)) = *x*

<u>Task: Simplify this function</u> (example):

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

By De Morgan's rule:

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

Boolean algebra

Commutative: *x* And *y* = *y* And *x*
x Or *y* = *y* Or *x*

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

Distributive: *x* And (*y* Or *z*) = (*x* And *y*) Or (*x* And *z*)
x Or (*y* And *z*)) = (*x* Or *y*) And (*x* Or *z*)

De Morgan: Not(*x* And *y*) = Not(*x*) Or Not(*y*)
Not(*x* Or *y*) = Not(*x*) And Not(*y*)

Idempotent: $\boxed{x \text{ And } x = x}$
x Or *x* = *x*

Double negation: Not(Not(*x*)) = *x*

<u>Task: Simplify this function</u> (example):

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

By De Morgan's rule:

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

By the associative rule:

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

Boolean algebra

Commutative:	x And y = y And x x Or y = y Or x
Associative:	x And (y And z) = (x And y) And z x Or (y Or z) = (x Or y) Or z
Distributive:	x And (y Or z) = (x And y) Or (x And z) x Or (y And z)) = (x Or y) And (x Or z)
De Morgan:	Not(x And y) = Not(x) Or Not(y) Not(x Or y) = $\boxed{\text{Not}(x) \text{ And Not}(y)}$
Idempotent:	x And x = x x Or x = x
Double negation:	Not(Not(x)) = x

<u>Task: Simplify this function</u> (example):

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

By De Morgan's rule:

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

By the associative rule:

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

By the idempotent rule:

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

Boolean algebra

Commutative: x And y = y And x
x Or y = y Or x

Associative: x And (y And z) = (x And y) And z
x Or (y Or z) = (x Or y) Or z

Distributive: x And (y Or z) = (x And y) Or (x And z)
x Or (y And z)) = (x Or y) And (x Or z)

De Morgan: Not(x And y) = Not(x) Or Not(y)
Not(x Or y) = Not(x) And Not(y)

Idempotent: x And x = x
x Or x = x

Double negation: $\boxed{\text{Not(Not}(x)) = x}$

<u>Task: Simplify this function</u> (example):

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

By De Morgan's rule:

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

By the associative rule:

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

By the idempotent rule:

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

By De Morgan's rule:

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

Boolean algebra

Commutative: x And y = y And x

x Or y = y Or x

Associative: x And (y And z) = (x And y) And z

x Or (y Or z) = (x Or y) Or z

Distributive: x And (y Or z) = (x And y) Or (x And z)

x Or (y And z)) = (x Or y) And (x Or z)

De Morgan: Not(x And y) = Not(x) Or Not(y)

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

Idempotent: x And x = x

x Or x = x

Double negation: Not(Not(x)) = x

<u>Task: Simplify this function</u> (example):

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

By De Morgan's rule:

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

By the associative rule:

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

By the idempotent rule:

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

By De Morgan's rule:

Not(Not(x Or y)) =

By double negation:

x* Or *y

Boolean algebra

Observations about simplifying Boolean functions:

- Can lead to significant optimization
- Based on intuition, experience, and luck
- Can be assisted by some tools
- But, in general: *NP*-hard.

Task: Simplify this function (example):

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

By De Morgan's rule:

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

By the associative rule:

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

By the idempotent rule:

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

By De Morgan's rule:

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

By double negation:

x* Or *y

Boolean algebra

Commutative: $(x \text{ And } y) = (y \text{ And } x)$
$(x \text{ Or } y) = (y \text{ Or } x)$

Associative: $(x \text{ And } (y \text{ And } z)) = ((x \text{ And } y) \text{ And } z)$
$(x \text{ Or } (y \text{ Or } z)) = ((x \text{ Or } y) \text{ Or } z)$

Distributive: $(x \text{ And } (y \text{ Or } z)) = (x \text{ And } y) \text{ Or } (x \text{ And } z)$
$(x \text{ Or } (y \text{ And } z)) = (x \text{ Or } y) \text{ And } (x \text{ Or } z)$

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

Idempotent: $(x \text{ And } x) = x$
$(x \text{ Or } x) = x$

Double negation: $\text{Not}(\text{Not}(x)) = x$

<u>Another example</u>: Prove that

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

Boolean algebra

Commutative: $(x \text{ And } y) = (y \text{ And } x)$

$(x \text{ Or } y) = (y \text{ Or } x)$

Associative: $(x \text{ And } (y \text{ And } z)) = ((x \text{ And } y) \text{ And } z)$

$(x \text{ Or } (y \text{ Or } z)) = ((x \text{ Or } y) \text{ Or } z)$

Distributive: $(x \text{ And } (y \text{ Or } z)) = (x \text{ And } y) \text{ Or } (x \text{ And } z)$

$(x \text{ Or } (y \text{ And } z)) = (x \text{ Or } y) \text{ And } (x \text{ Or } z)$

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

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

Idempotent: $(x \text{ And } x) = x$

$(x \text{ Or } x) = x$

Double negation: $\text{Not}(\text{Not}(x)) = x$

<u>Another example</u>: Prove that

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

De Morgan:

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

Negate both sides:

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

By double negation:

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

<u>One conclusion:</u>

- We don’t really “need” Or
- We will soon revisit this reduction

Chapter 1: Boolean logic

Theory

- Basic concepts
- Boolean algebra
- Boolean functions
- Nand

Practice

- Logic gates
- HDL
- Hardware simulation
- Multi-bit buses

Project 1

- Introduction
- Chips
- Guidelines

Boolean function

<u>Formula</u> (example)

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

<u>Truth table</u>

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

A Boolean function can be expressed using a *formula*, or a *truth table*

The two representations are equivalent

Question: Can we construct each representation from the other one?

Formula ➡ truth table

Formula

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

Formula ➡ truth table

Formula

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

Truth table

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	

Formula ➡ truth table

Formula

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

Truth table

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

When $y = 0$, f must be 0

Formula ➡ truth table

<u>Formula</u>

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

<u>Truth table</u>

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

When $y = 0$, f must be 0

When $x = 0$, f must be 0

Formula ➡ truth table

Formula

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

Truth table

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

When $y = 0$, f must be 0

When $x = 0$, f must be 0

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

(1 And (Not(1) Or 1)) And 1 = 1

Formula ➡ truth table

Formula

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

Truth table

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

Lemma

Given a Boolean function expressed as a formula, we can always construct from it its truth table.

Proof: Evaluate the function over all the possible values of its variables (which is *the* definition of a truth table)

Boolean function synthesis: Truth table → formula

Formula

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

?

Truth table

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

Can we also go the other way around, *for any given truth table*?

Boolean function synthesis: Truth table ➡ formula

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

Goal: Synthesize a formula which is equivalent to this truth table

Boolean function synthesis

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

1. Focus on the rows where $f = 1$

Boolean function synthesis

x	y	z	f	f_1	f_3	f_5
0	0	0	1	1	0	0
0	0	1	0	0	0	0
0	1	0	1	0	1	0
0	1	1	0	0	0	0
1	0	0	1	0	0	1
1	0	1	0	0	0	0
1	1	0	0	0	0	0
1	1	1	0	0	0	0

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

Not(x) And y And Not(z)

x And Not(y) And Not(z)

1. Focus on the rows where $f = 1$
2. For each such row i, define a function f_i which equals 1 in row i and 0 elsewhere. Define f_i to be the conjunction of all of the variables or their negations, as the variable's value is 1 or 0 in the i'th row

Boolean function synthesis

x	y	z	f	f_1	f_3	f_5
0	0	0	1	1	0	0
0	0	1	0	0	0	0
0	1	0	1	0	1	0
0	1	1	0	0	0	0
1	0	0	1	0	0	1
1	0	1	0	0	0	0
1	1	0	0	0	0	0
1	1	1	0	0	0	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))

1. Focus on the rows where $f = 1$
2. For each such row i, define a function f_i which equals 1 in row i and 0 elsewhere. Define f_i to be the conjunction of all of the variables or their negations, as the variable's value is 1 or 0 in the i'th row
3. Define f to be the disjunction of all these conjunctions

Boolean function synthesis

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))

Disjunctive Normal Form (DNF)

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

following simplification

1. Focus on the rows where $f = 1$
2. For each such row i, define a function f_i which equals 1 in row i and 0 elsewhere. Define f_i to be the conjunction of all of the variables or their negations, as the variable's value is 1 or 0 in the i'th row
3. Define f to be the disjunction of all these conjunctions

Boolean function synthesis

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

<u>Lemma</u>

For any boolean function expressed as a truth table, we can synthesize a formula that expresses that function

Proof: Use the truth table to construct the DNF (which is a formula)

<u>Theorem</u>

Any boolean function can be represented as a formula containing only the operators And, Or, Not

Proof: Construct the function's truth table, then use the truth table to construct the DNF (which, by definition, uses only And, Or, Not).

Chapter 1: Boolean logic

Theory

- ✓ Basic concepts
- ✓ Boolean algebra
- ✓ Boolean functions
- ➡ Nand

Practice

- Logic gates
- HDL
- Hardware simulation
- Multi-bit buses

Project 1

- Introduction
- Chips
- Guidelines

The expressive power of Nand

x Nand y

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

x And y

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

x Or y

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

Not(x)

x	Not
0	1
1	0

Observations

- Not(x) = x Nand x
- x And y = Not$(x$ Nand $y)$
- x Or y = Not(Not(x) And Not(y))
  (De Morgan)

Thus:

- Not can be realized using Nand
- And can be realized using Nand
- Or can be realized using Nand

Theorem: Any Boolean function can be realized using only Nand.

Proof : Any Boolean function can be expressed using Not, And, and Or (DNF).
Combined with the above observations, we get the theorem.

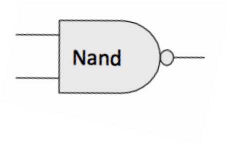
The expressive power of Nand

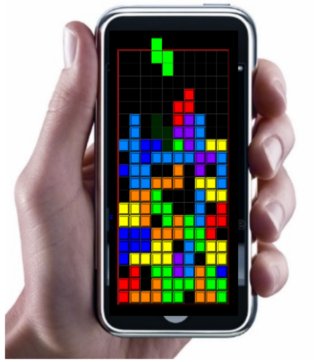
<u>Theorem:</u> Any Boolean function can be realized using only Nand.

<u>Conclusion:</u> Any computer can be built from Nand gates only:

x Nand *y*

x	*y*	Nand
0	0	1
0	1	1
1	0	1
1	1	0





<u>Computers:</u>
Machines that realize Boolean functions:

f (input bits) = output bits

OK, so we *can* build a computer from Nand gates only.

But… how can we *actually* do it?

That's what the Nand to Tetris course is all about!

Chapter 1: Boolean logic

Theory

- Basic concepts
- Boolean algebra
- Boolean functions
- Nand

Practice

- Logic gates
- HDL
- Hardware simulation
- Multi-bit buses

Project 1

- Introduction
- Chips
- Guidelines

Chapter 1: Boolean logic

Theory

- Basic concepts
- Boolean algebra
- Boolean functions
- Nand

Practice

- Logic gates
- HDL
- Hardware simulation
- Multi-bit buses

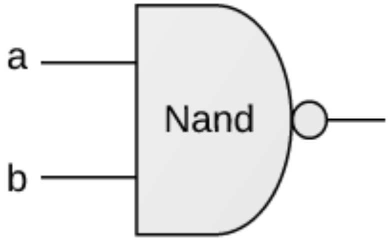
Project 1

- Introduction
- Chips
- Guidelines

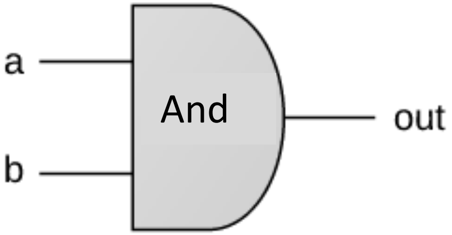
Logic gates

- Elementary gates (`Nand`, `And`, `Or`, `Not`, …)
- Composite gates (`Mux`, `Adder`, …)

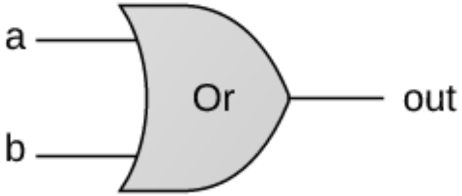
Elementary gates



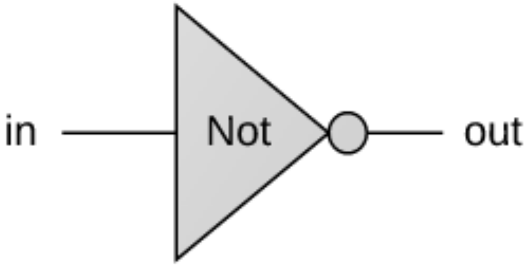

```
 if (a==1 and b==1)
then out=0 else out=1
```




```
 if (a==1 and b==1)
then out=1 else out=0
```




```
 if (a==1 or b==1)
then out=1 else out=0
```

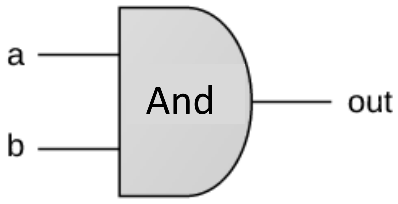



```
      if (in==0)
then out=1 else out=0
```

<u>Why focus on these particular gates?</u>

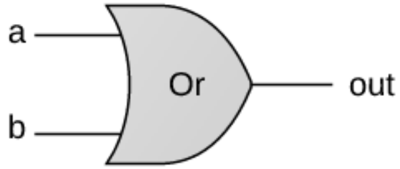
- Because either {`Nand`} or {`And`, `Or`, `Not`} (as well as other subsets) can be used to span any given Boolean function
- Because they have efficient hardware implementations.

Elementary gates



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

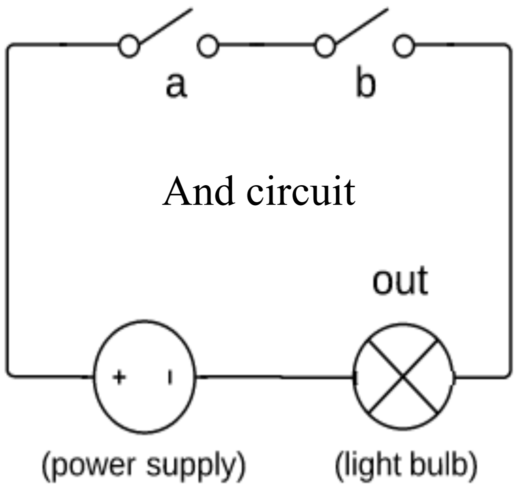
```
if (a==1 and b==1)
then out=1 else out=0
```

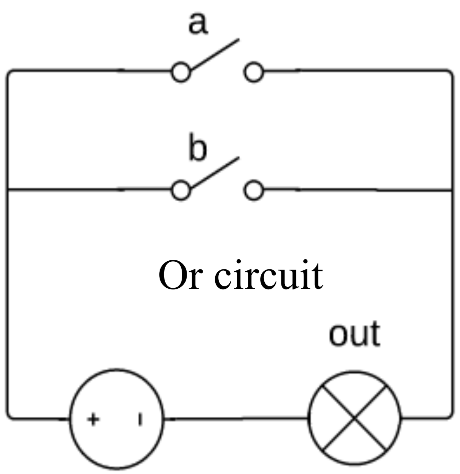


x	*y*	Or
0	0	0
0	1	1
1	0	1
1	1	1

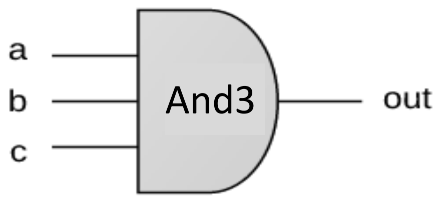
```
if (a==1 or b==1)
then out=1 else out=0
```

<u>Circuit implementations</u> (conceptual):





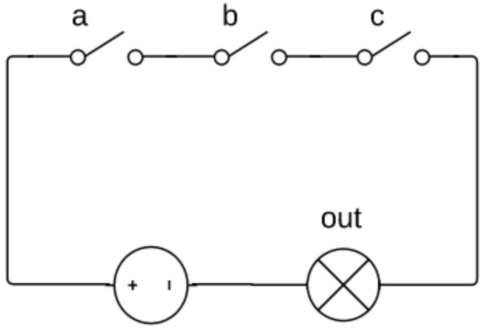
Composite gates



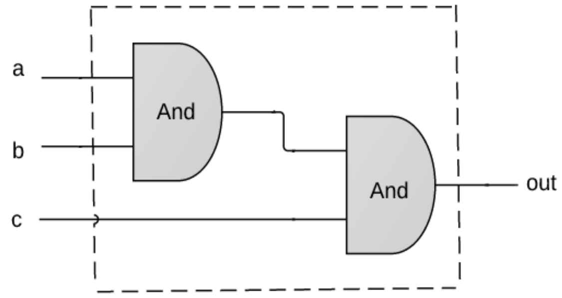

```
if (a==1 and b==1 and c==1)
   then out=1 else out=0
```

Possible implementations:

Physical



Logical



- This course does not deal with physical implementations
  (circuits, transistors,... that's EE, not CS)
- We'll focus on logical implementations.

Chapter 1: Boolean logic

Theory

- Basic concepts
- Boolean algebra
- Boolean functions
- Nand

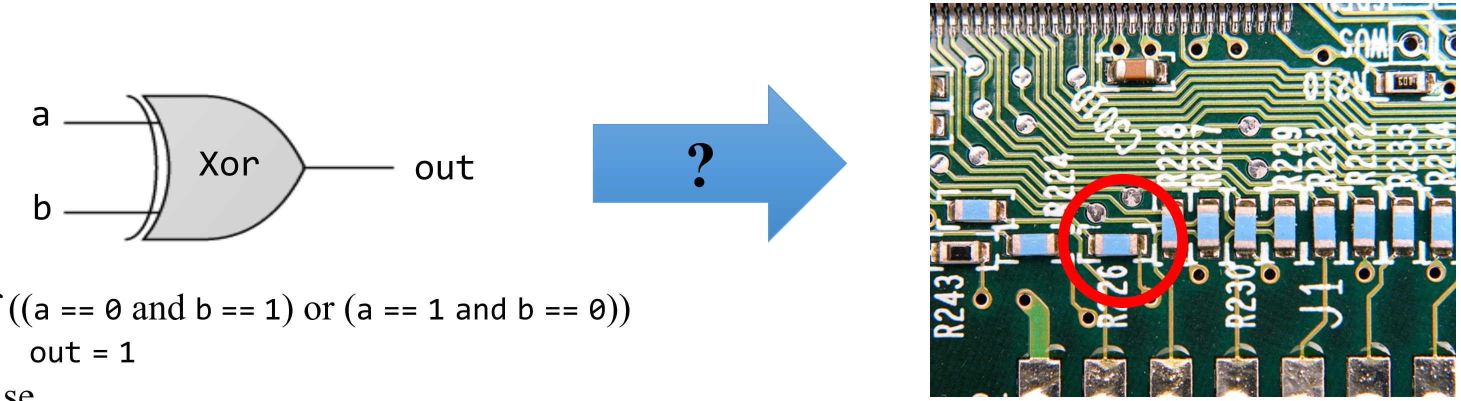
Practice

- ✓ Logic gates
- HDL
- Hardware simulation
- Multi-bit buses

Project 1

- Introduction
- Chips
- Guidelines

Building a chip

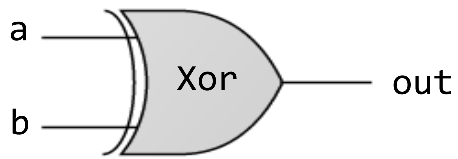


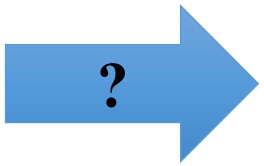

```
if ((a == 0 and b == 1) or (a == 1 and b == 0))
    out = 1
else
    out = 0
```

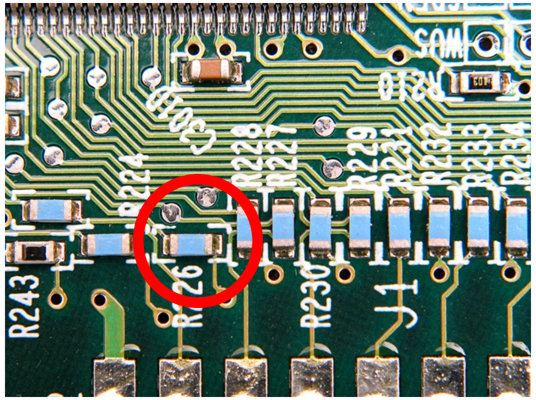
<u>The process</u>

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

Building a chip





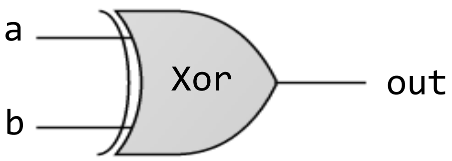


```
if ((a == 0 and b == 1) or (a == 1 and b == 0))
    out = 1
else
    out = 0
```

<u>The process</u>

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

Design: Requirements



if ((`a == 0` and `b == 1`) or (`a == 1` and `b == 0`))
    `out = 1`
else
    `out = 1`

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

Requirement

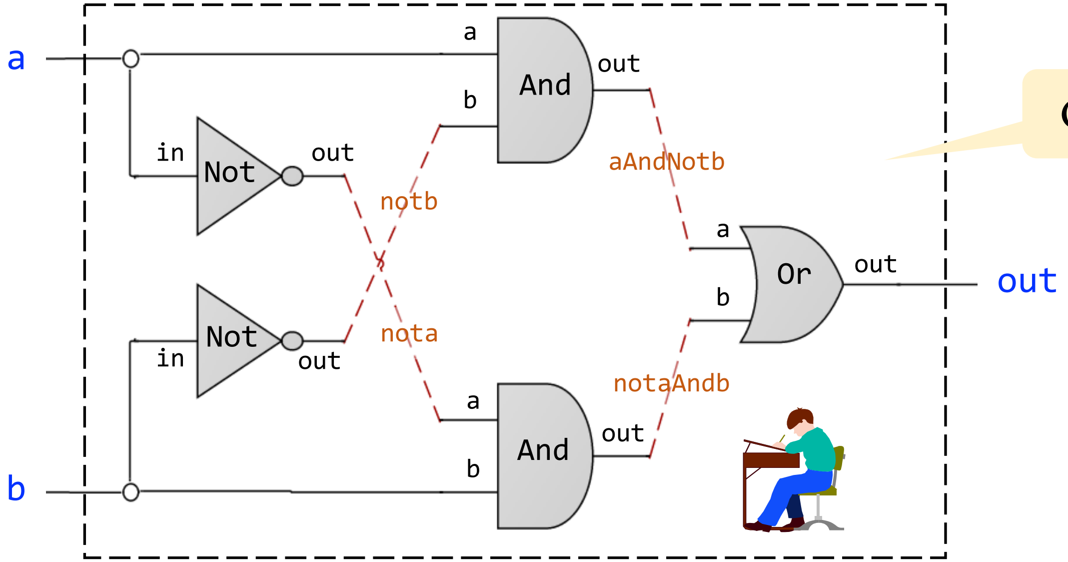
Build a chip that delivers this functionality

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

HDL *stub file*

```
/** Chips API: */
...
Not (in= , out= );
And (a= , b= , out= );
Or  (a= , b= , out= );
Xor (a= , b= , out= );
...
```

Design: Implementation



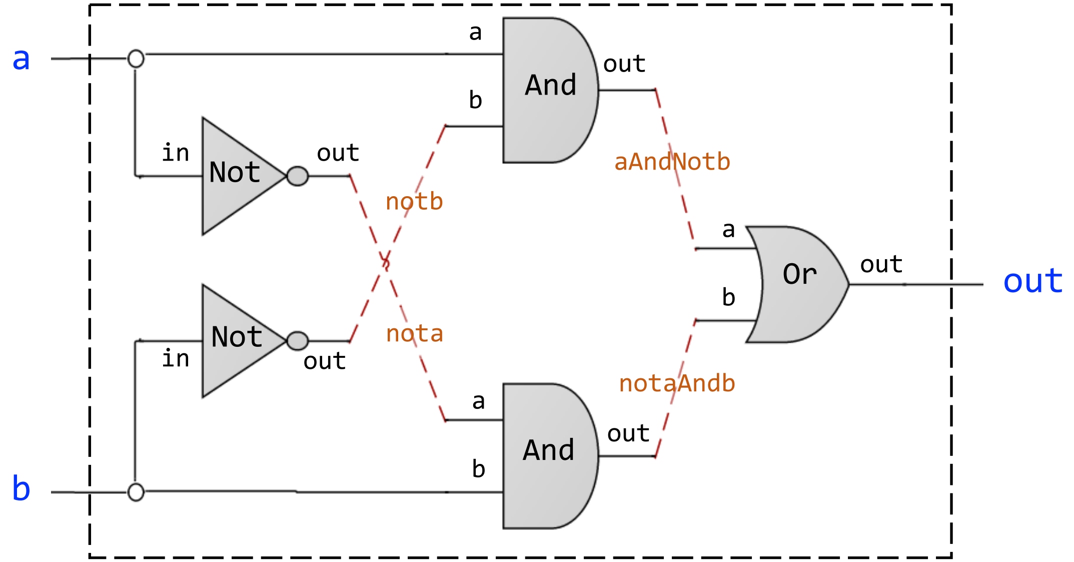
Gate diagram

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

HDL *stub file*

```
/** Chips API: */
...
Not (in= , out= );
And (a= , b= , out= );
Or  (a= , b= , out= );
Xor (a= , b= , out= );
...
```

Design: Implementation

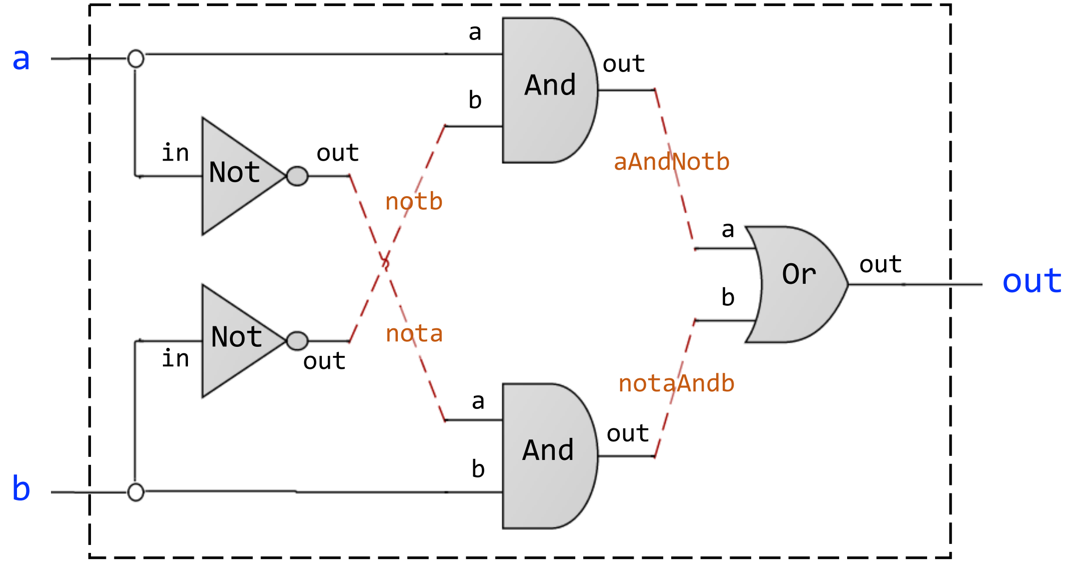



```
/** Sets  out = (a And Not(b)) Or (Not(a) And b))  */
CHIP Xor {
    IN a, b;
    OUT out;

    PARTS:
    // Missing implementation
```

HDL *stub file*

```
/** Chips API: */
...
Not (in= , out= );
And (a= , b= , out= );
Or  (a= , b= , out= );
Xor (a= , b= , out= );
...
```

Design: Implementation

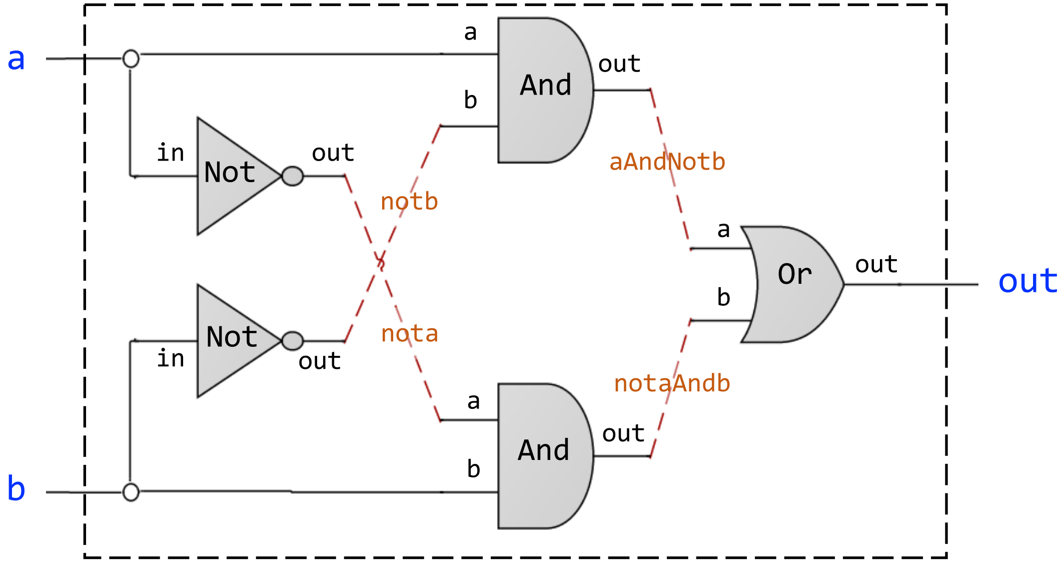



```
/** Sets  out = (a And Not(b)) Or (Not(a) And b))  */
CHIP Xor {
    IN a, b;
    OUT out;

    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);

}
```

```
/** Chips API: */
...
Not (in= , out= );
And (a= , b= , out= );
Or  (a= , b= , out= );
Xor (a= , b= , out= );
...
```

Practice: Complete the missing HDL code

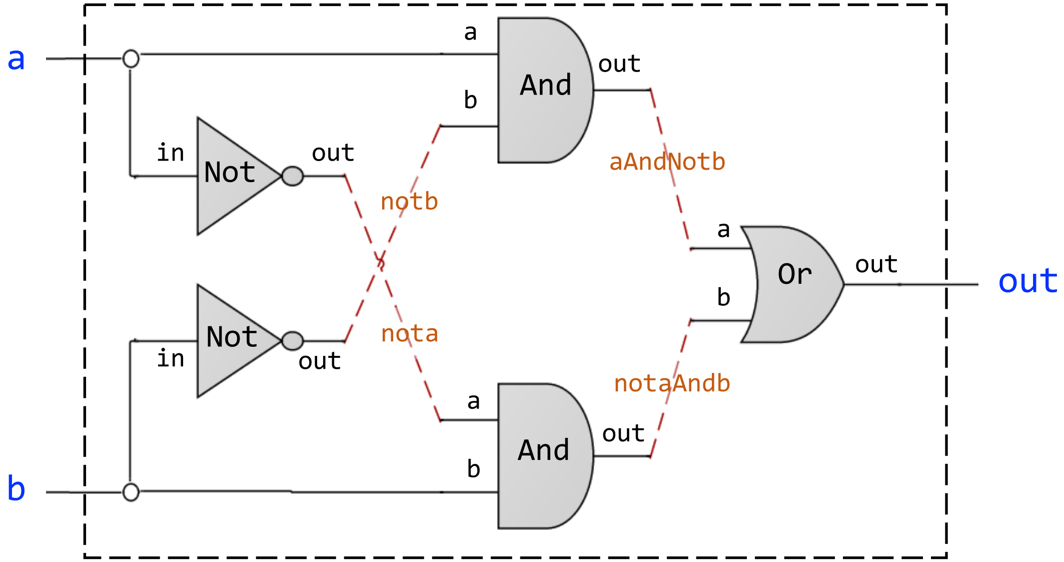
Design: Implementation




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

```
/** Chips API: */
...
Not (in= , out= );
And (a= , b= , out= );
Or  (a= , b= , out= );
Xor (a= , b= , out= );
...
```

Interface / Implementation



gate interface

gate Implement-ation

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

<u>A logic gate has:</u>

- One interface
- Many possible implementations

Hardware description languages

Observations:

- HDL: a functional / declarative language
- An HDL program can be viewed as a textual specification of a chip diagram
- The order of HDL statements is insignificant.

```
/** Sets  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 description languages

Common HDLs

- VHDL
- Verilog
- …

Our HDL

- Captures the essence of other HDLs
- Minimal and simple
- Provides all you need for this course

Our HDL Guide / Documentation:

[*The Elements of Computing Systems* / Appendix 2: HDL]

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

Chapter 1: Boolean logic

Theory

- Basic concepts
- Boolean algebra
- Boolean functions
- Nand

Practice

- ✓ Logic gates
- ✓ HDL
- ➡ Hardware simulation
- Multi-bit buses

Project 1

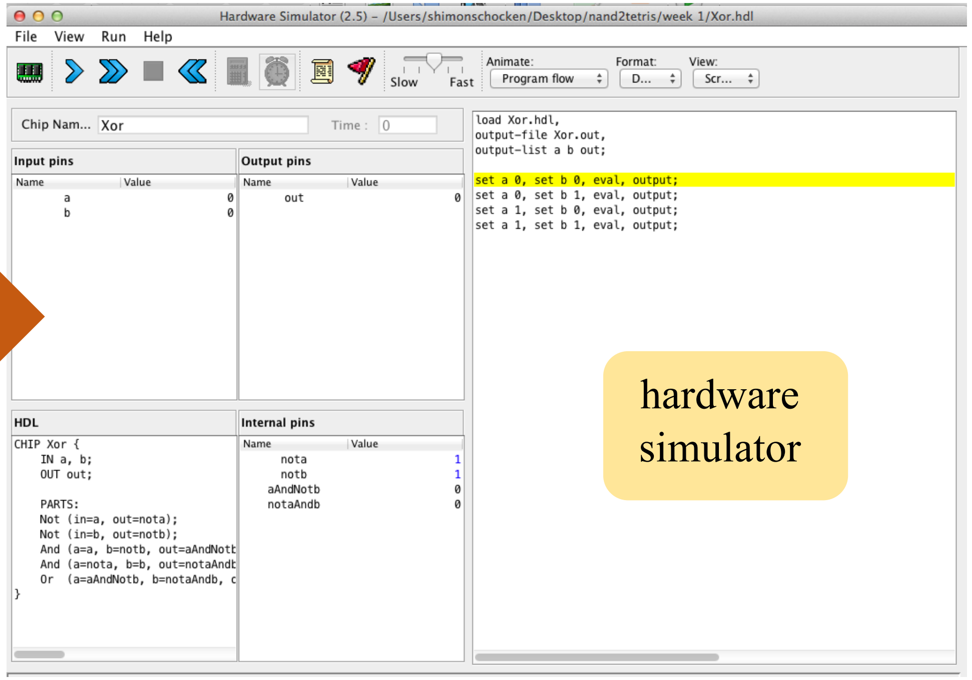
- Introduction
- Chips
- Guidelines

Hardware simulation in a nutshell

```
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 code

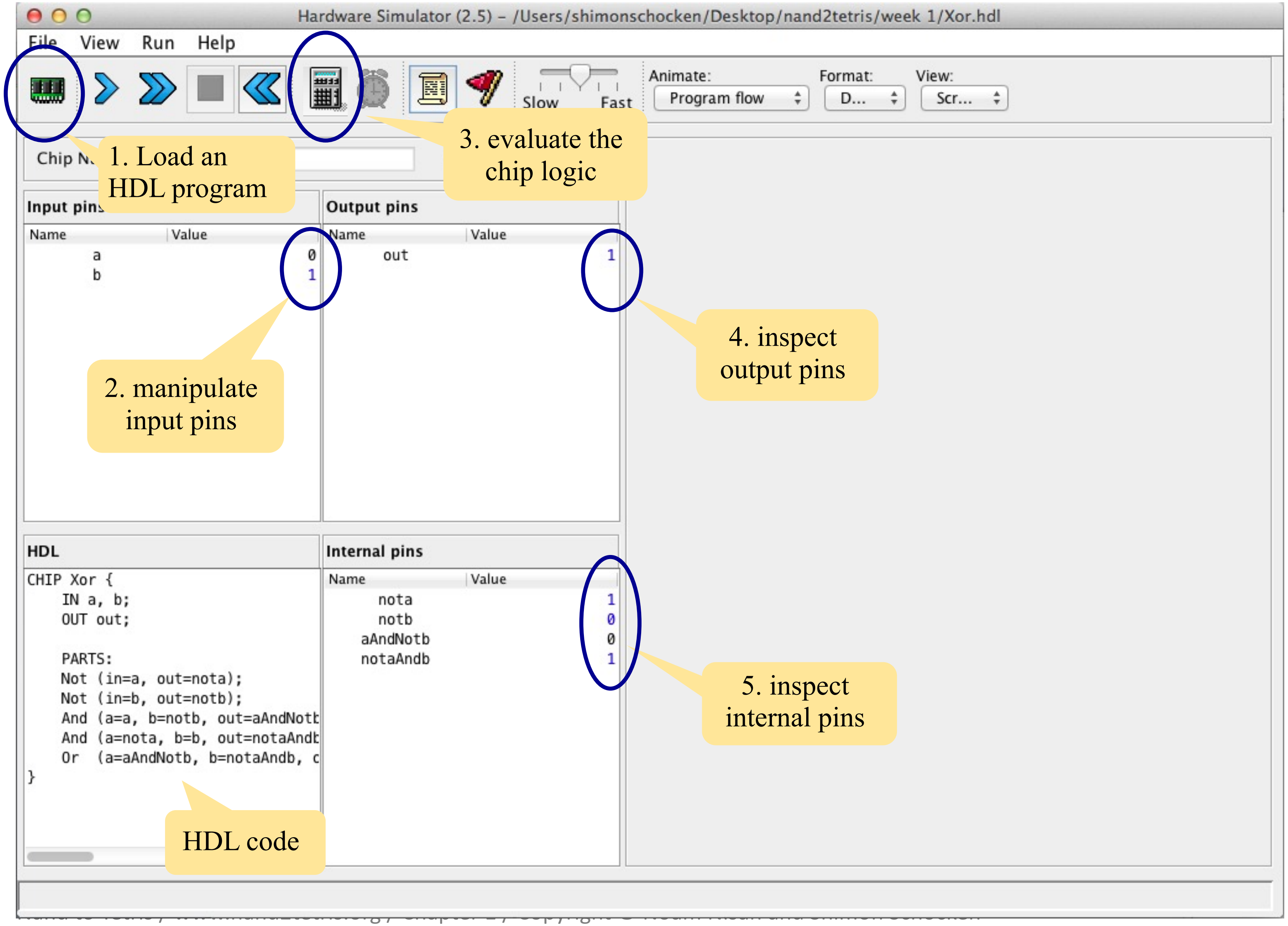
simulate



Simulation options

- Interactive
- Script-based.

Interactive simulation



Hardware simulation in a nutshell

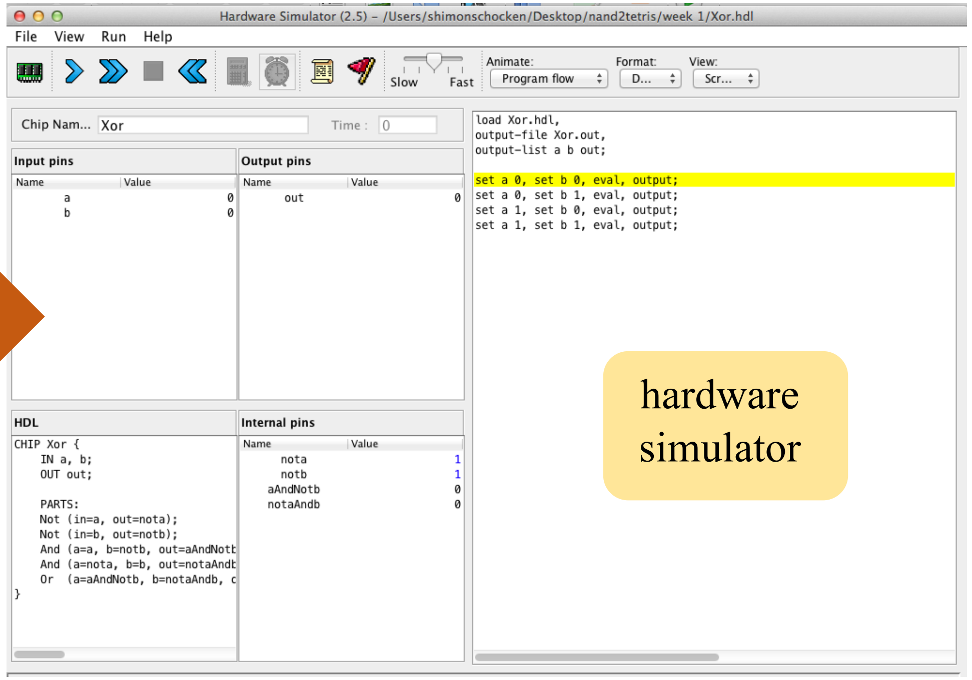
HDL code

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

    PARTS:
    Not (in=a, out=nota);
    Not (in=b, out=notb);
    And
    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



<u>Simulation options</u>

- ✓ Interactive
- ➡ Script-based.

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);
}
```

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 = sequence 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);
}
```

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

Xor.out

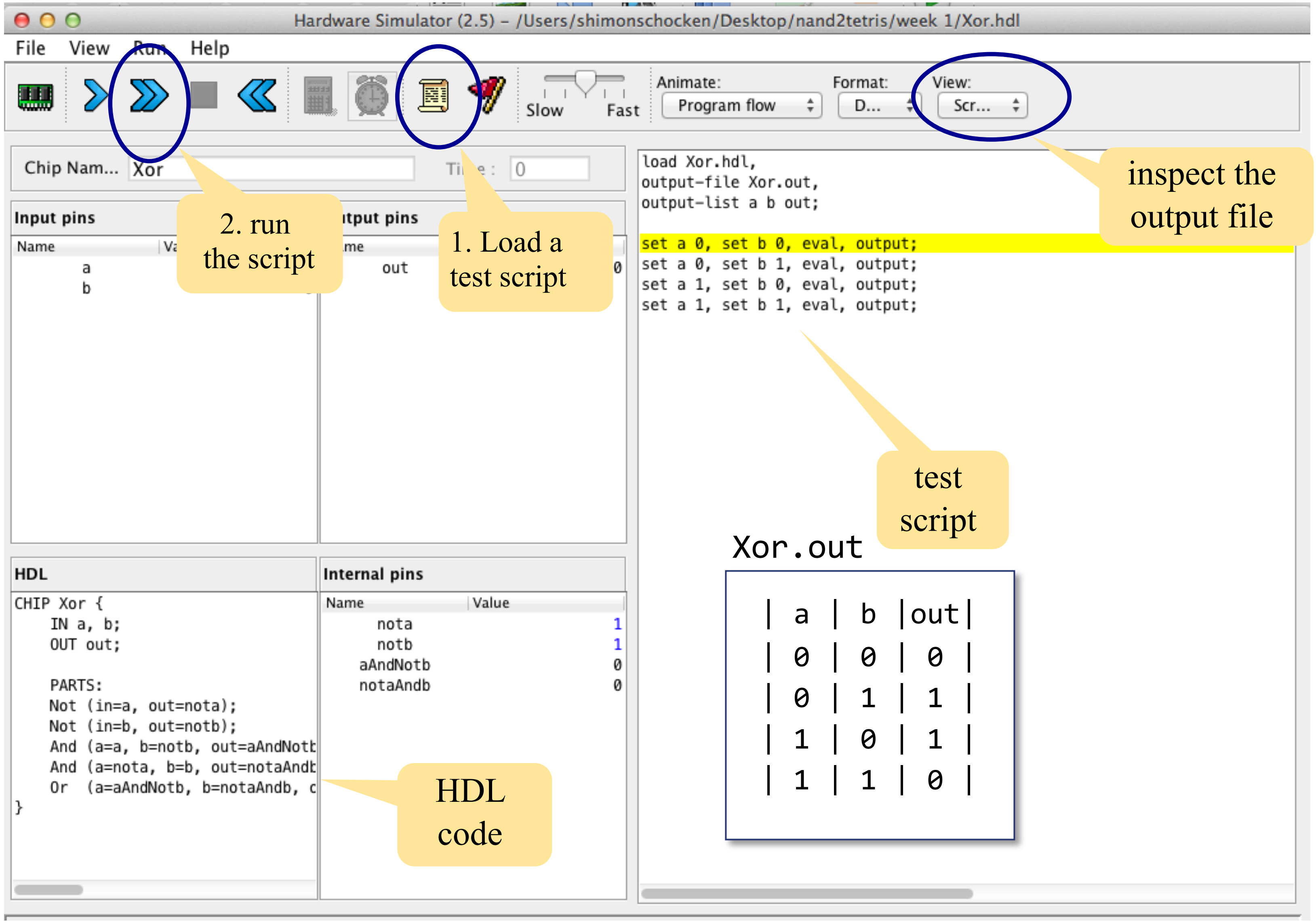
```
| a | b |out|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
```

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

The logic of a typical test script

- Initialize:
    - Loads an HDL file
    - Creates an empty output file
    - Lists the names of the pins whose values will be written to the output file
- Repeat:
    - `set – eval - output`

Script-based 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);
}
```

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;
```

Xor.out

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

Script-based simulation, with a compare 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);
}
```

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;
```

Xor.out

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

compare

Xor.cmp

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

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.

Script-based simulation, with a compare 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);
}
```

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;
```

Xor.out

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

compare

Xor.cmp

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

Demos:

[Experimenting with Built-In Chips]

[Building and Testing HDL-based Chips]

[Script-Based Chip Testing]

Chapter 1: Boolean logic

Theory

- Basic concepts
- Boolean algebra
- Boolean functions
- Nand

Practice

- ✓ Logic gates
- ✓ HDL
- ✓ Hardware simulation
- ➡ Multi-bit buses

Project 1

- Introduction
- Chips
- Guidelines

Multi-bit bus

- Sometimes we wish to manipulate a *sequence of bits* as a single entity
- Such a multi-bit entity is termed "bus"

Example: 16-bit bus

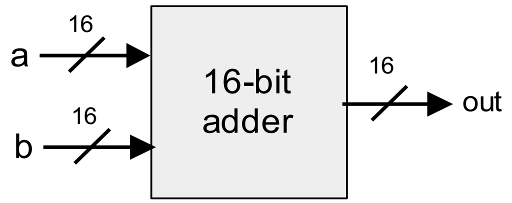
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	0	0	0	1	1	0	1	1	1	0	1

MSB = Most significant bit

LSB = Least significant bit

Working with buses: Example

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

   PARTS:
   ...
}
```

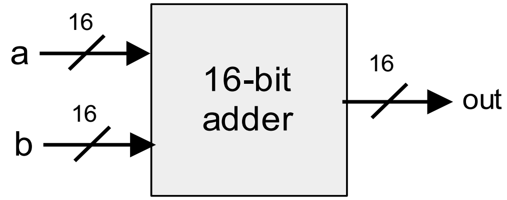


	15	…	1	0
a:	1	…	1	1
b:	0	…	1	0
c:	0	…	0	1
out:	1	…	1	0

```
/* Adds three 16-bit inputs. */
CHIP Adder3Way {
   IN a[16], b[16], c[16];
   OUT out[16];

   PARTS:
   Adder(a=  , b= , out=   );
   Adder(a=  , b= , out=   );
}
```

Working with buses: Example

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

   PARTS:
   ...
}
```



	15	…	1	0
a:	1	…	1	1
b:	0	…	1	0
c:	0	…	0	1
out:	1	…	1	0

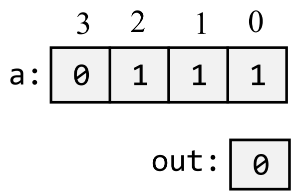
```
/* Adds three 16-bit inputs. */
CHIP Adder3Way {
   IN a[16], b[16], c[16];
   OUT out[16];

   PARTS:
   Adder(a=a , b=b, out=ab);
   Adder(a=ab, b=c, out=out);
}
```

n-bit value (bus) can be treated as a single entity

Creates an internal bus pin (ab)

Working with individual bits within buses

```
/* Returns 1 if a==1 and b==1,
   0 otherwise. */
CHIP And {
   IN a, b;
   OUT out;
   ...
}
```




```
/* 4-way And: Ands 4 bits. */
CHIP And4Way {
   IN a[4];
   OUT out;

   PARTS:
   And(a=      , b=     , out=      );
   And(a=      , b=     , out=      );
   And(a=      , b=     , out=      );
}
```

Working with individual bits within buses

```
/* Returns 1 if a==1 and b==1,
   0 otherwise. */
CHIP And {
   IN a, b;
   OUT out;
   ...
}
```

```
    3   2   1   0
a: [0] [1] [1] [1]

       out: [0]
```

```
/* 4-way And: Ands 4 bits. */
CHIP And4Way {
   IN a[4];
   OUT out;

   PARTS:
   And(a=a[0],    b=a[1], out=and01);
   And(a=and01,   b=a[2], out=and012);
   And(a=and012,  b=a[3], out=out);
}
```

Input bus pins can be subscripted.

Working with individual bits within buses

```
/* Returns 1 if a==1 and b==1,
   0 otherwise. */
CHIP And {
   IN a, b;
   OUT out;
   ...
}
```

	3	2	1	0
a:	0	1	1	1
out:				0

```
/* 4-way And: Ands 4 bits. */
CHIP And4Way {
   IN a[4];
   OUT out;

   PARTS:
   And(a=a[0],   b=a[1], out=and01);
   And(a=and01,  b=a[2], out=and012);
   And(a=and012, b=a[3], out=out);
}
```

Input bus pins can be subscripted.

	3	2	1	0
a:	0	1	0	1
b:	0	0	1	1
out:	0	0	0	1

```
/* Bit-wise And of two 4-bit inputs */
CHIP And4 {
   IN a[4], b[4];
   OUT out[4];
```

Working with individual bits within buses

```
/* Returns 1 if a==1 and b==1,
   0 otherwise. */
CHIP And {
   IN a, b;
   OUT out;
   ...
}
```

	3	2	1	0
a:	0	1	1	1

out:	0

```
/* 4-way And: Ands 4 bits. */
CHIP And4Way {
   IN a[4];
   OUT out;

   PARTS:
   And(a=a[0],    b=a[1], out=and01);
   And(a=and01,   b=a[2], out=and012);
   And(a=and012,  b=a[3], out=out);
}
```

Input bus pins can be subscripted.

	3	2	1	0
a:	0	1	0	1
b:	0	0	1	1
out:	0	0	0	1

```
/* Bit-wise And of two 4-bit inputs */
CHIP And4 {
   IN a[4], b[4];
   OUT out[4];

   PARTS:
   And(a=    , b=    , out=      );
   And(a=    , b=    , out=      );
   And(a=    , b=    , out=.     );
   And(a=    , b=    , out=      );
}
```

Working with individual bits within buses

```
/* Returns 1 if a==1 and b==1,
   0 otherwise. */
CHIP And {
   IN a, b;
   OUT out;
   ...
}
```

	3	2	1	0
a:	0	1	1	1

out: 0

```
/* 4-way And: Ands 4 bits. */
CHIP And4Way {
   IN a[4];
   OUT out;

   PARTS:
   And(a=a[0],   b=a[1], out=and01);
   And(a=and01,  b=a[2], out=and012);
   And(a=and012, b=a[3], out=out);
}
```

Input bus pins can be subscripted.

	3	2	1	0
a:	0	1	0	1
b:	0	0	1	1
out:	0	0	0	1

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

Output bus pins can be subscripted

More on busses:

[*The Elements of Computing Systems* / Appendix 2: HDL]

Chapter 1: Boolean logic

Theory

- Basic concepts
- Boolean algebra
- Boolean functions
- Nand

Practice

- Logic gates
- HDL
- Hardware simulation
- Multi-bit buses

Project 1

- Introduction
- Chips
- Guidelines

Chapter 1: Boolean logic

Theory

- Basic concepts
- Boolean algebra
- Boolean functions
- Nand

Practice

- Logic gates
- HDL
- Hardware simulation
- Multi-bit buses

Project 1

- Introduction
- Chips
- Guidelines

Built-in chips

We provide built-in versions of the chips built in this course (in `tools/builtInChips`).
For example:

`Xor.hdl`

```
/** Sets out to a Xor 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);
}
```

Implemented in HDL

`Xor.hdl`

```
/** Sets out to a Xor b */
CHIP Xor {
    IN a, b;
    OUT out;

    BUILTIN Xor;
    // implemented by a Xor.java class.
}
```

Implemented in Java

A built-in chip has the same interface as the regular chip, but a different implementation

Behavioral simulation

- Before building a chip in HDL, one can implement the chip logic in a high-level language
- Enables experimenting with / testing the chip abstraction before actually building it
- Enables high-level planning and testing of hardware architectures.

Demo: Loading and testing a built-in chip in the hardrawe simulator

Hardware construction projects

<u>Key players:</u>

- Architect:
  - Decides which chips are needed
  - Specifies the chips
- Developers:
  - Build / test the chips

<u>In Nand to Tetris:</u>

The architect is the course team; the developers are the students

For each chip, the architect supplies:

- Built-in chip
- Chip API (skeletal HDL program = stub file)
- Test script
- Compare file

Given these resources, the developers (students) build the chips.

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

Xor.hdl

```
/** Sets out to a Xor 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

compare file

Xor.cmp

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

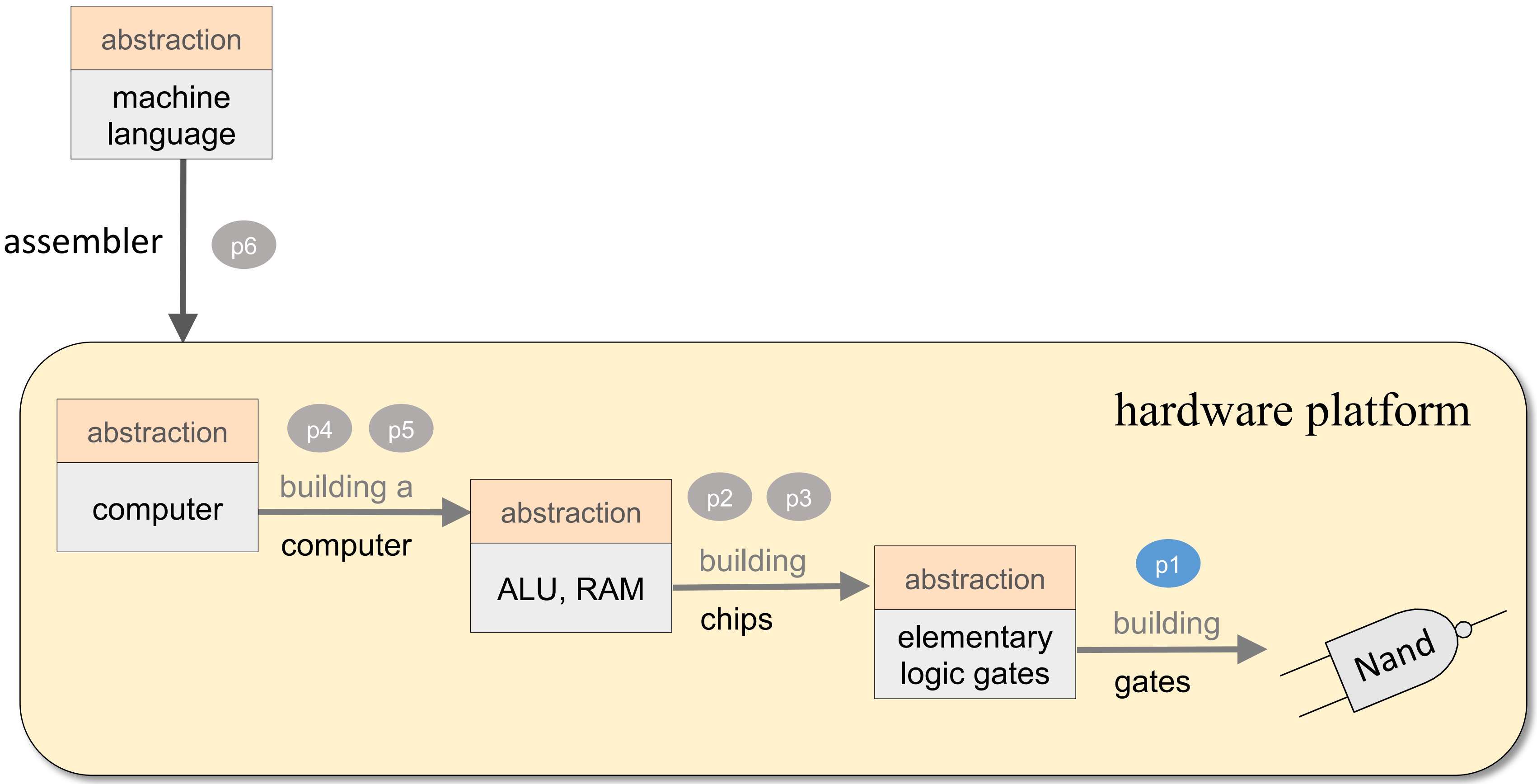
These files specify:

- The chip interface (.hdl)
- How the chip is supposed to behave (.cmp)
- How to test the chip (.tst)

The developer's task:

Implement the chip (complete the supplied .hdl file), using these resources.

Project 1



<u>Project 1</u>

Build 15 elementary logic gates

Project 1

Given: `Nand`

Goal: Build the following gates:

Elementary logic gates	16-bit variants	Multi-way variants
`Not`	`Not16`	`Or8Way`
`And`	`And16`	`Mux4Way16`
`Or`	`Or16`	`Mux8Way16`
`Xor`	`Mux16`	`DMux4Way`
`Mux`		`DMux8Way`
`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:

Elementary logic gates

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

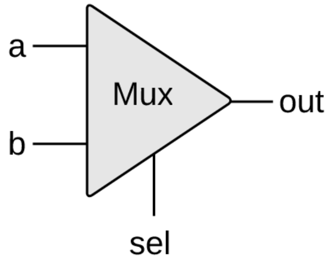
16-bit variants

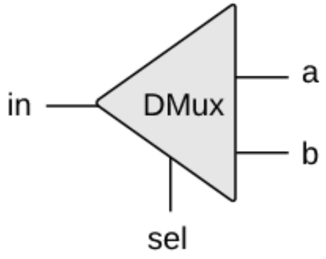
- `Not16`
- `And16`
- `Or16`
- `Mux16`

Multi-way variants

- `Or8Way`
- `Mux4Way16`
- `Mux8Way16`
- `DMux4Way`
- `DMux8Way`

Multiplexor / Demultiplexor





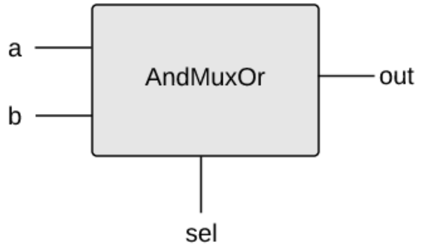

```
if (sel == 0)
    out = a
else
    out = b
```

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

<u>Widely used in:</u>

- Hardware design
- Communications networks.

Example 1: Using Mux logic to build a programmable gate

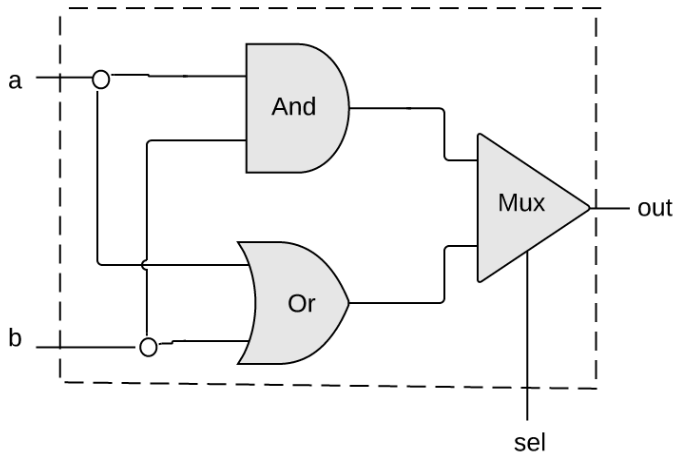


if (sel == 0)
    out = a And b
else
    out = a Or b

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

When sel == 0
the gate acts like an And gate

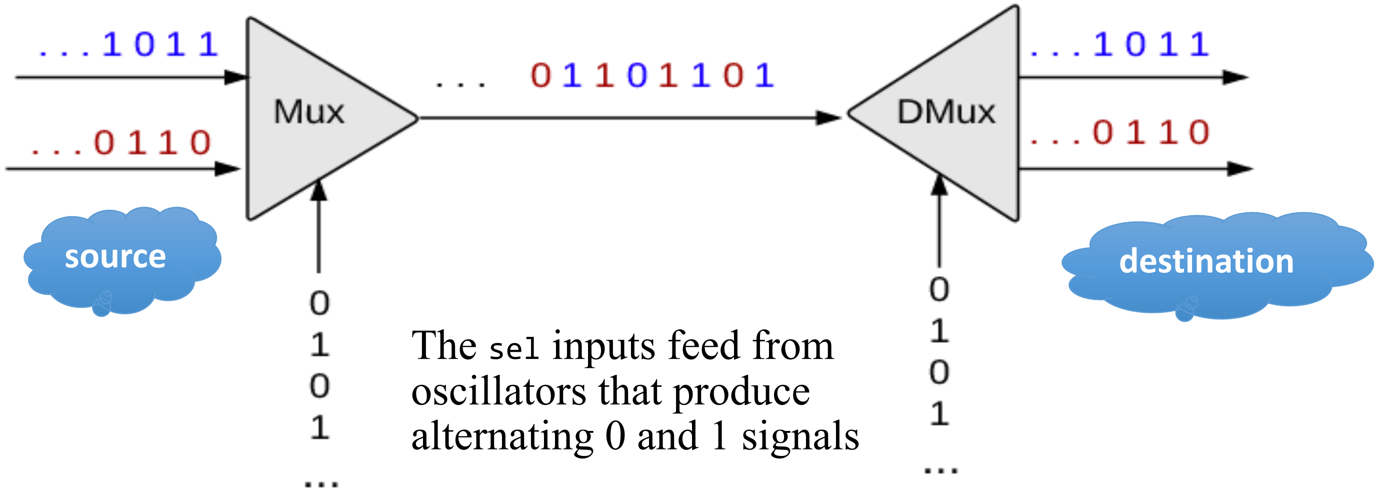
When sel == 1
the gate acts like an Or gate



Mux.hdl

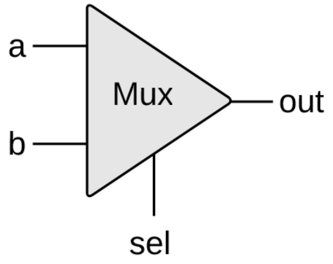
```
CHIP AndMuxOr {
    IN a, b, sel;
    OUT out;

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

Example 2: Using Mux logic to build an interleaved channel



- Enables transmitting multiple messages simultaneously using a single, shared communications line
- *Conceptual, and unrelated to this course.*

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

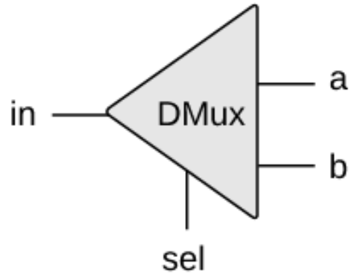
Mux.hdl

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

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

Implementation tip

Can be implemented from And, Or, Not.

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
- Routes the single input value to one of two possible destinations

DMux.hdl

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

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

<u>Implementation tip</u>

Similar to the `Mux` implementation.

Project 1

Elementary logic gates

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

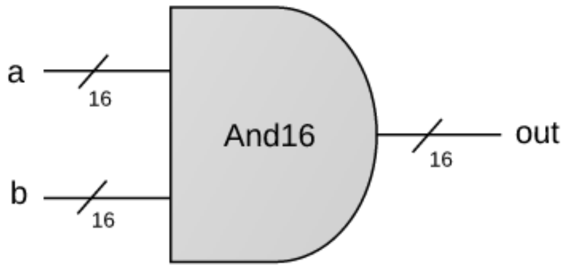
16-bit variants

- Not16
- And16
- Or16
- Mux16

Multi-way variants

- Or8Way
- Mux4Way16
- Mux8Way16
- DMux4Way
- DMux8Way

And16



Example:

```
  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
      -------------------------------
out = 0 0 1 0 1 0 0 1 0 0 0 0 1 0 0 0
```

```
CHIP And16 {
    IN a[16], b[16];
    OUT out[16];

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

<u>Implementation tip</u>

A straightforward 16-bit extension of the elementary `And` gate

(See the HDL documentation about working with *multi-bit buses*).

Project 1

<u>Elementary logic gates</u>

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

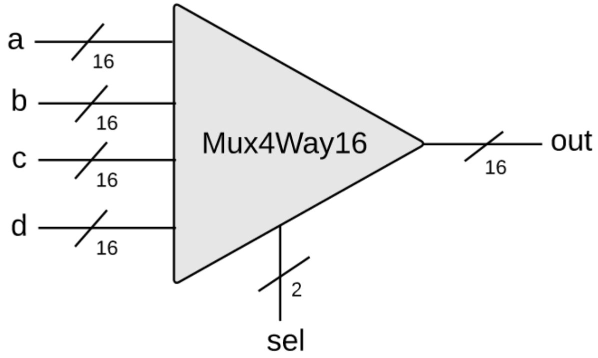
<u>16-bit variants</u>

- `Not16`
- `And16`
- `Or16`
- `Mux16`

<u>Multi-way variants</u>

- `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:
}
```

<u>Implementation tip:</u>

Can be built from several `Mux16` gates.

Chapter 1: Boolean logic

Theory

- Basic concepts
- Boolean algebra
- Boolean functions
- Nand

Practice

- Logic gates
- HDL
- Hardware simulation
- Multi-bit buses

Project 1

✓ Introduction

✓ Chips

➡ Guidelines

Project 1

Elementary logic gates

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

16-bit variants

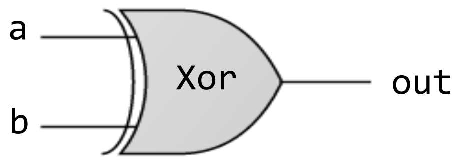
- Not16
- And16
- Or16
- Mux16

Multi-way variants

- Or8Way
- Mux4Way16
- Mux8Way16
- DMux4Way
- DMux8Way

How to actually build these gates?

Files



if ((a == 0 and b == 1) or
(a == 1 and b == 0))
sets out = 1
else
sets out = 0

Xor.cmp

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

For every chip built in the course (using Xor as an example), we supply these three files

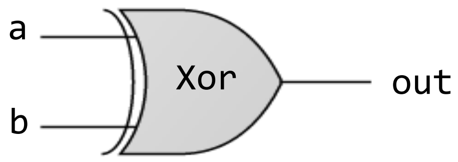
Xor.hdl (stub file)

```
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;
```

Files



if ((a == 0 and b == 1) or
(a == 1 and b == 0))
sets out = 1
else
sets out = 0

`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` (stub file)

```
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 folder

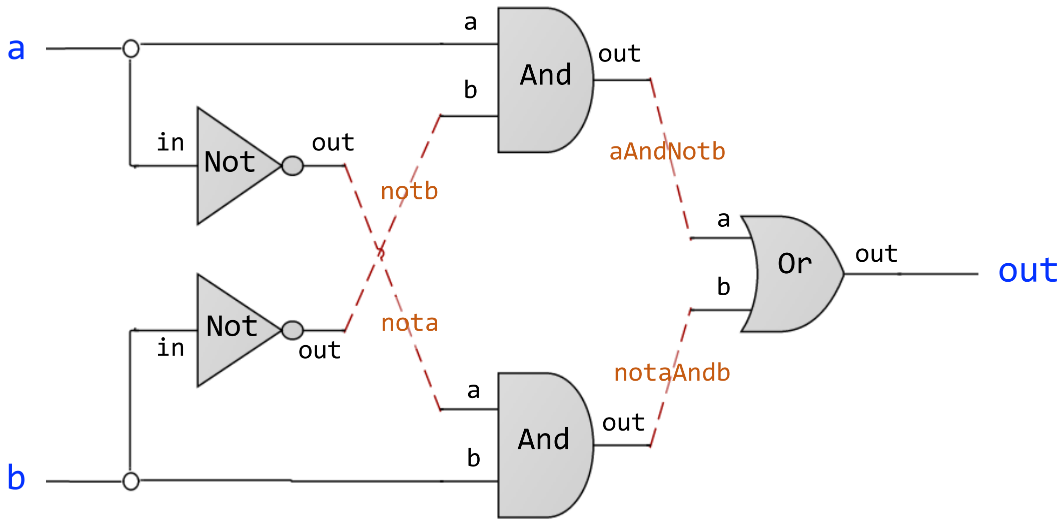
(`.hdl`, `.tst`, `.cmp` files):

`nand2tetris/projects/01`

Tools:

- Text editor
  (for completing the `.hdl` files)
- Hardware simulator:
  `nand2tetris/tools`

Chip interfaces




```
CHIP Xor {
    IN a, b;
    OUT out;
    PARTS:
    Not (in= , out= );
    Not (in= , out= );
    And (a= , b=, out=);
    And (a= , b=, out=);
    Or  (a= , b=, out=);
}
```

If I want to use some chip-parts, how do I figure out their signatures?

Chip interfaces: Hack chip set API

Open the Hack chip set API in a window, and copy-paste chip signatures into your HDL code, as needed

```
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= ,writeM= ,ad
DFF (in= ,out= );
DMux4Way (in= ,sel= ,a= ,b= ,c= ,d= );
DMux8Way (in= ,sel= ,a= ,b= ,c= ,d= ,e= ,f= ,g= ,h=
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= ,h= ,sel= ,ou
```

```
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= );
```

Built-in chips

```
CHIP Foo {
    IN ...;
    OUT ...;

    PARTS:
    ...
    Bar(...)
    ...
}
```

Q: How can I play with / use a chip-part before implementing it?

A: The simulator features built-in chip implementations

Forcing the simulator to use a built-in chip, say `Bar`:

- Typically, `Bar.hdl` will be either a given stub-file, or a file that has an incomplete implementation
- Remove, or rename, the file `Bar.hdl` from the project folder
- Whenever `Bar` will be mentioned as a chip-part in some chip definition, the simulator will fail to find `Bar.hdl` in the current folder. This will cause the simulator to invoke the built-in version of `Bar` instead.

Project 1

Guidelines: www.nand2tetris.org (projects section)

Files: `nand2tetris/projects/01` (on your PC)

Tools

- Text editor (for completing the given `.hdl` stub-files)
- Hardware simulator: `nand2tetris/tools` (on your PC)

Guides

- Hardware simulator tutorial
- HDL guide
- Hack chip set API

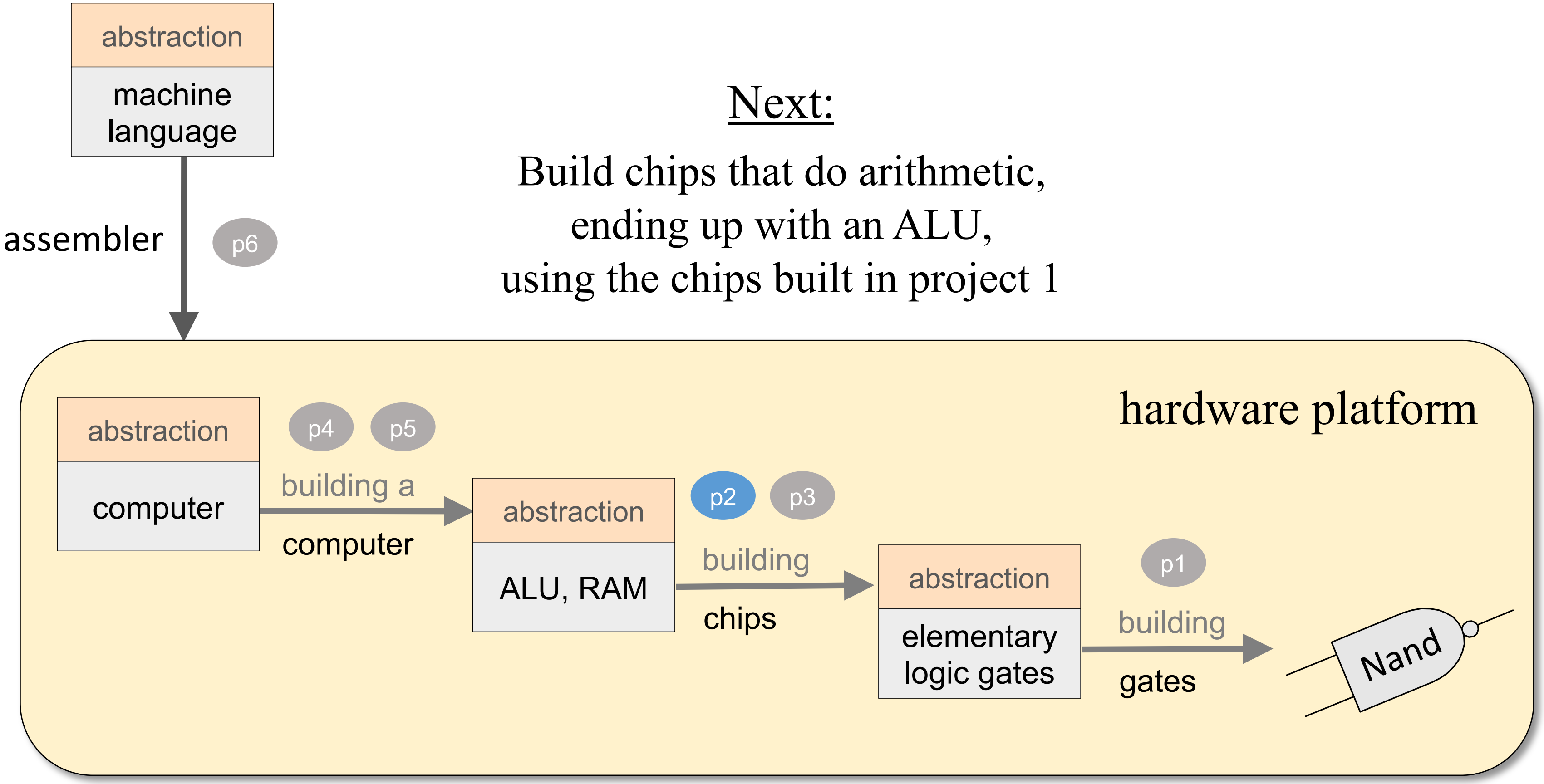
Best practice advice

- Implement the chips in the order in which they appear in the project guidelines
- If you don't implement some chips, you can still use them as chip-parts in other chips (use their built-in implementations)
- You can invent additional, "helper chips"; However, this is not necessary. Implement and use only the chips that the architects (we) specified
- In each chip implementation, strive to use as few chip-parts as possible
- When defining 16-bit chips, the same chip-parts may appear many times. That's fine, use copy-paste-edit.

That's It!

Go Do Project 1!

What's next?



Next:

Build chips that do arithmetic,
ending up with an ALU,
using the chips built in project 1

This project / chapter:

Build 15 elementary logic gates