



## Lecture 1

# Boolean Logic

Slide deck for

“Chapter 1: Boolean Logic” of the book

*The Elements of Computing Systems*

By Noam Nisan and Shimon Schocken

MIT Press, 2nd edition

# Chapter 1: Boolean logic

---

## Theory



Basic concepts

- Nand

## Practice

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

## Project 1

- Introduction
- Chips
- Guidelines

# Boolean values

---



# Boolean values

---



# Boolean values

---



off      no      false      0



on      yes      true      1



George Boole  
1815 - 1864

Different labels, all referring to two possible states.

# Boolean values

---

$b_1$     $b_0$



1 binary variable: 2 possible states

# Boolean values

---

$b_1$     $b_0$



1 binary variable: 2 possible states

2 binary variables: 4 possible states

# Boolean values

---

$b_2$     $b_1$     $b_0$



1 binary variable: 2 possible states

2 binary variables: 4 possible states



# Boolean values

---

$\dots b_2 \quad b_1 \quad b_0$



1 binary variable: 2 possible states

2 binary variables: 4 possible states

3 binary variables: 8 possible states

$\dots$

# Boolean values

---

$\dots$   $b_2$   $b_1$   $b_0$



1 binary variable: 2 possible states

2 binary variables: 4 possible states

3 binary variables: 8 possible states

$\dots$

Question: How many different states can be represented by  $N$  binary variables?

# Boolean values

---

$\dots$   $b_2$   $b_1$   $b_0$



1 binary variable: 2 possible states

2 binary variables: 4 possible states

3 binary variables: 8 possible states













$\dots$

Question: How many different states can be represented by  $N$  binary variables?

Answer:  $2^N$

# Boolean functions

---

$x$	$y$	$f$
		
		
		
		

# Boolean functions

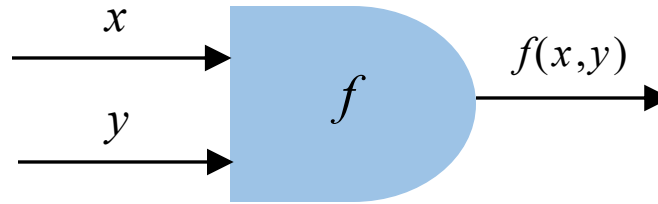
---

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

# Boolean functions

---

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

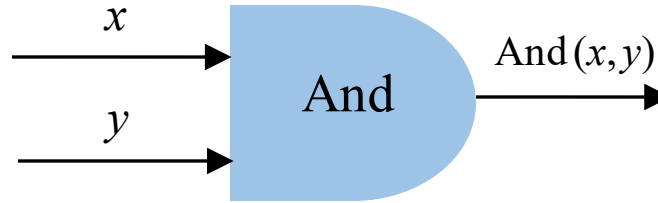


$$f(x,y) = \begin{cases} 1 & \text{when } x == 1 \text{ and } y == 1 \\ 0 & \text{otherwise} \end{cases}$$

# Boolean functions

---

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



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

## Boolean function

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

Simple boolean functions (like And):

- Sometimes called *operators*
- Their variables are called *operands*
- $f(x, y)$  can also be written as  $x \text{ } f \text{ } y$

# Boolean functions

---

$x$  And  $y$

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



# 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

# 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

# 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

# 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

# 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

• • •

$x$   $f$   $y$

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

• • •

$x f y$

$x$	$y$	$f$
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):

# 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

---

## Theory



Basic concepts



Nand

## Practice

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

## Project 1

- Introduction
- Chips
- Guidelines

# The expressive power of Nand

---

$x \text{ Nand } y$			$x \text{ And } y$			$x \text{ Or } y$			$\text{Not}(x)$	
$x$	$y$	Nand	$x$	$y$	And	$x$	$y$	Or	$x$	Not
0	0	1	0	0	0	0	0	0	0	1
0	1	1	0	1	0	0	1	1	1	0
1	0	1	1	0	0	1	0	1	0	1
1	1	0	1	1	1	1	1	1	1	0

## Observations

- $\text{Not}(x) = x \text{ Nand } x$
- $x \text{ And } y = \text{Not}(x \text{ Nand } y)$
- $x \text{ Or } y = \text{Not}(\text{Not}(x) \text{ And } \text{Not}(y))$   
(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 as a truth table. Any truth table can be expressed as a Boolean function using only Not, And, and Or (synthesized as a DNF, Disjunctive Normal Form). Combined with the above observations, Q.E.D.

# The expressive power of Nand

---

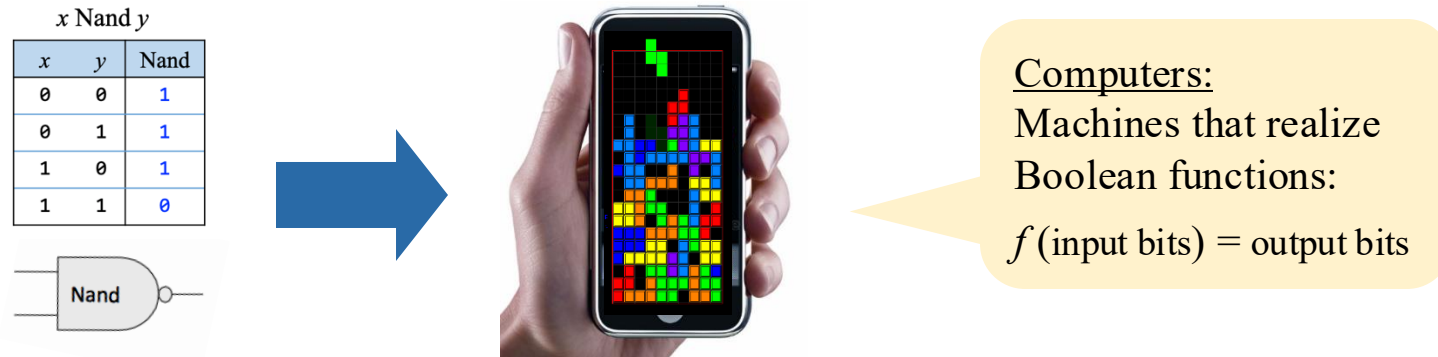
Theorem: Any Boolean function can be realized using only Nand.

# The expressive power of Nand

---

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

Implication: Any computer can be built from Nand gates only:



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
- Nand

## Practice

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

## Project 1

- Introduction
- Chips
- Guidelines

# Chapter 1: Boolean logic

---

## Theory

- Basic concepts
- Nand

## Practice



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

## Project 1

- Introduction
- Chips
- Guidelines

# Logic gates

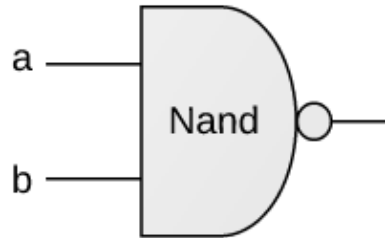
---

➡ Elementary gates (Nand, And, Or, Not, ...)

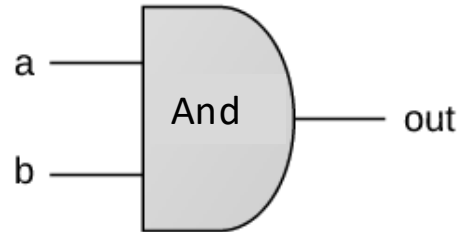
- Composite gates (Mux, Adder, ...)

# Elementary gates

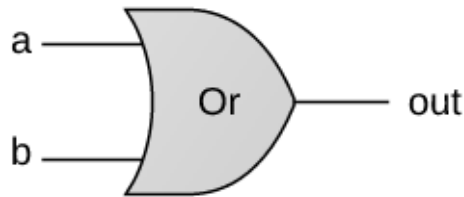
---



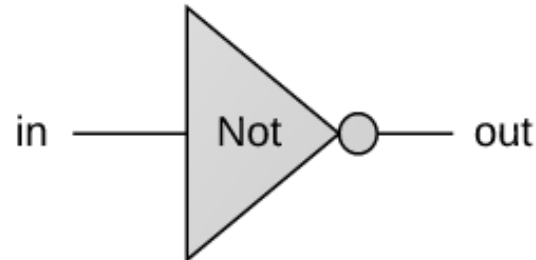
```
if (a==1 and b==1)
then out=0 else out=1
```



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



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



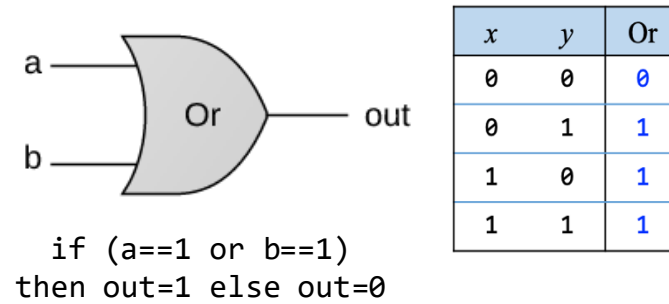
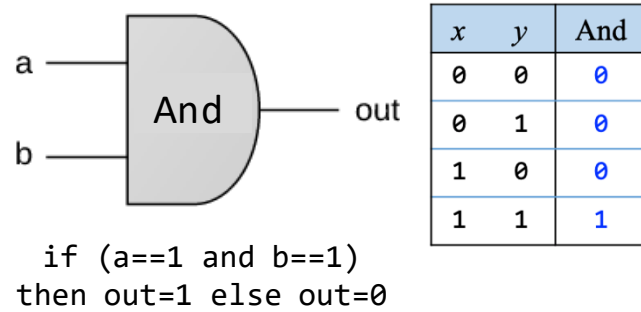
```
if (in==0)
then out=1 else out=0
```

## Why focus on these particular gates?

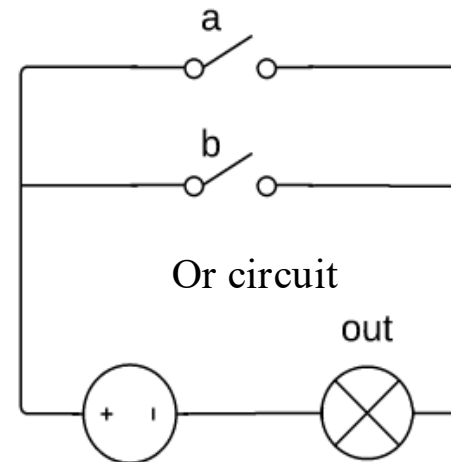
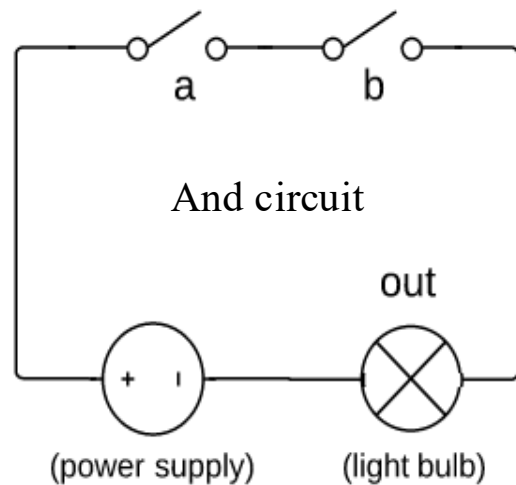
- Because either {Nand} or {And, Or, Not} (as well as other subsets of Boolean operators) can be used to span any given Boolean function
- Because they have efficient and direct hardware implementations.



# Elementary gates

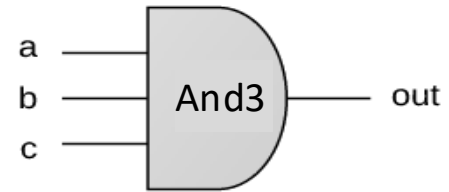


## Circuit implementations (conceptual):



# Composite gates

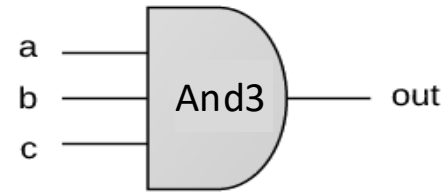
---



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

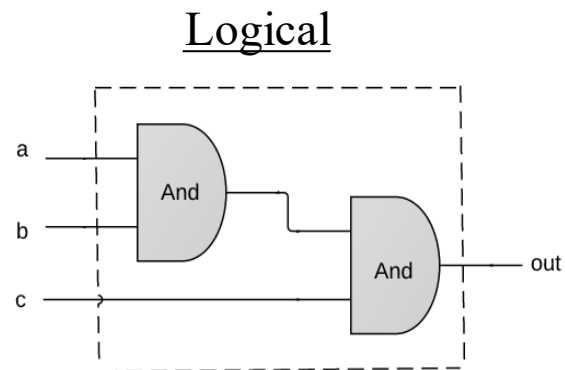
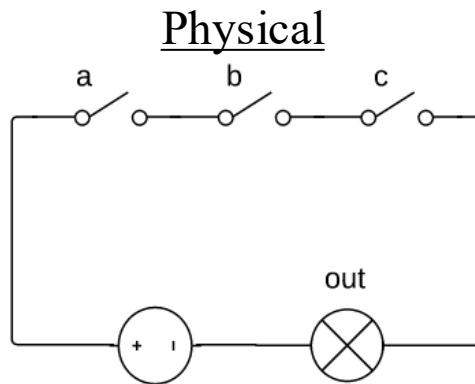
# Composite gates

---



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

Possible implementations:



- 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
- Nand

## Practice



Logic gates



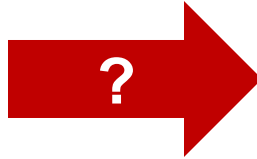
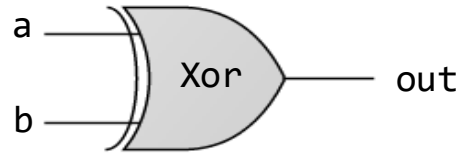
HDL

- Hardware simulation
- Multi-bit buses

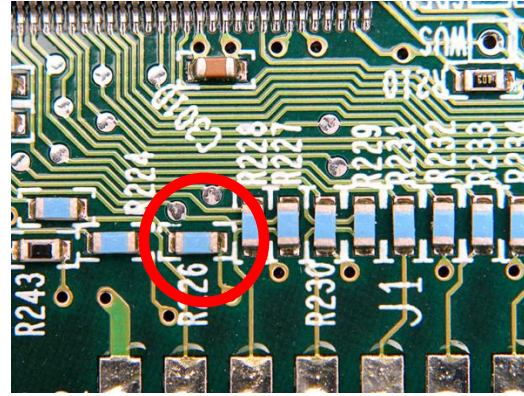
## Project 1

- Introduction
- Chips
- Guidelines

# Building a chip



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

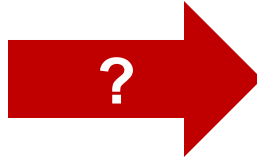
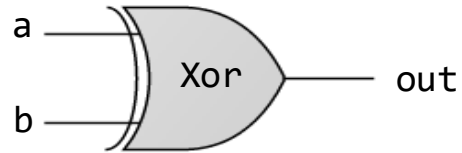


(an arbitrary image found on the Internet...)

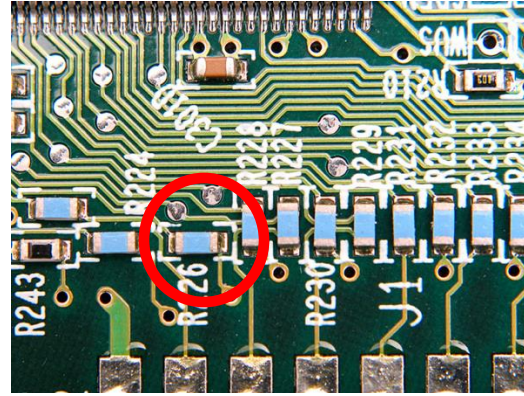
## The process

- 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
```

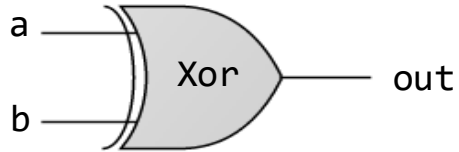


(an arbitrary image found on the Internet...)

## The process

- ✓ 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 = 0
```

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

## Requirement

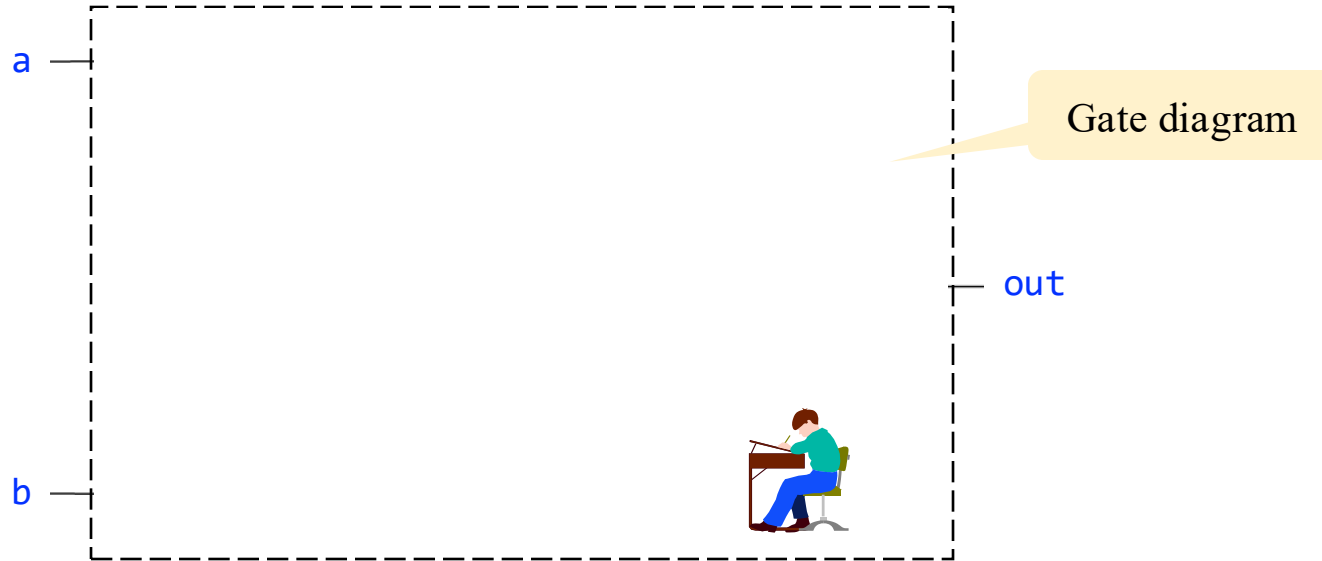
Build a chip that delivers this functionality

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

*HDL stub file*

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

# Design: Implementation



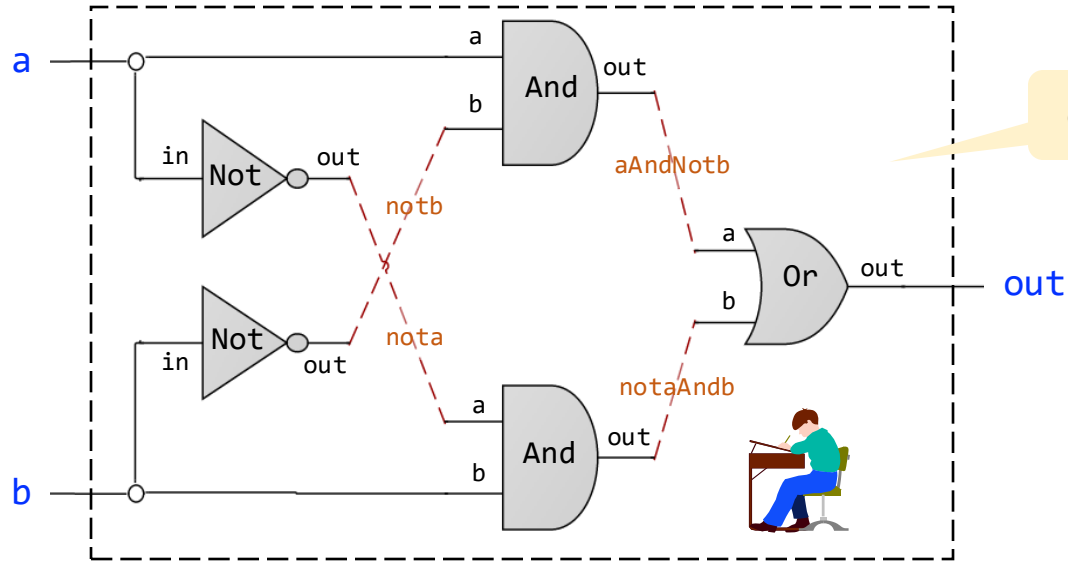
```
/** out = (a And Not(b)) Or (Not(a) And b) */  
CHIP Xor {  
  IN a, b;  
  OUT out;  
  PARTS:  
    // Missing implementation  
}
```

*HDL stub file*

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



# Design: Implementation

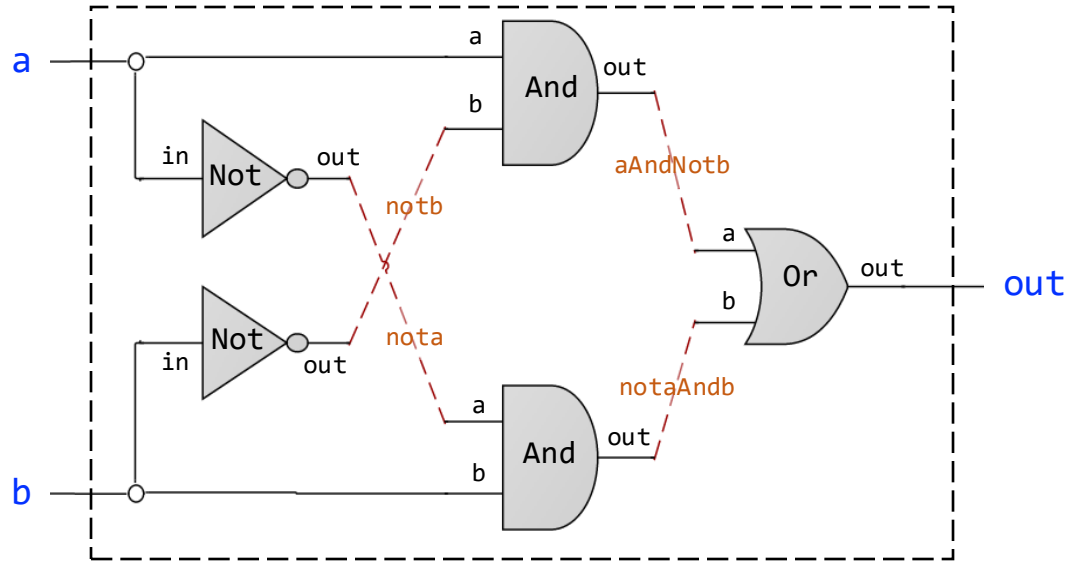


```
/** out = (a And Not(b)) Or (Not(a) And b) */  
CHIP Xor {  
  IN a, b;  
  OUT out;  
  PARTS:  
    // Missing implementation  
}
```

HDL stub file

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

# Design: Implementation



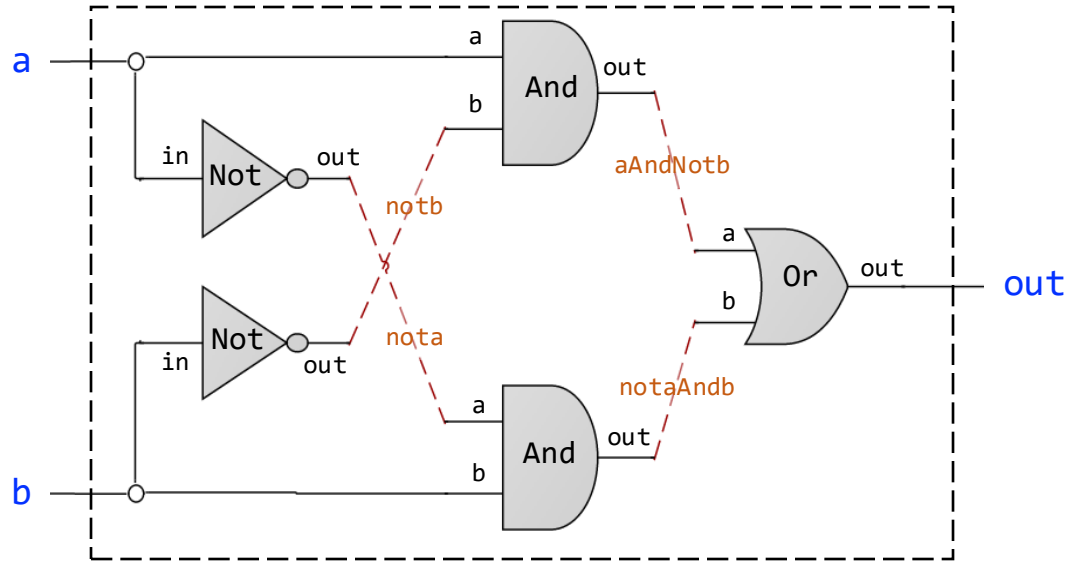
```
/** out = (a And Not(b)) Or (Not(a) And b) */  
CHIP Xor {  
  IN a, b;  
  OUT out;  
  PARTS:  
    // Missing implementation  
}
```



HDL stub file

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

# Design: Implementation



```
/** 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 set (APIs): */
```

```
...
```

```
Not (in=, out= );
```

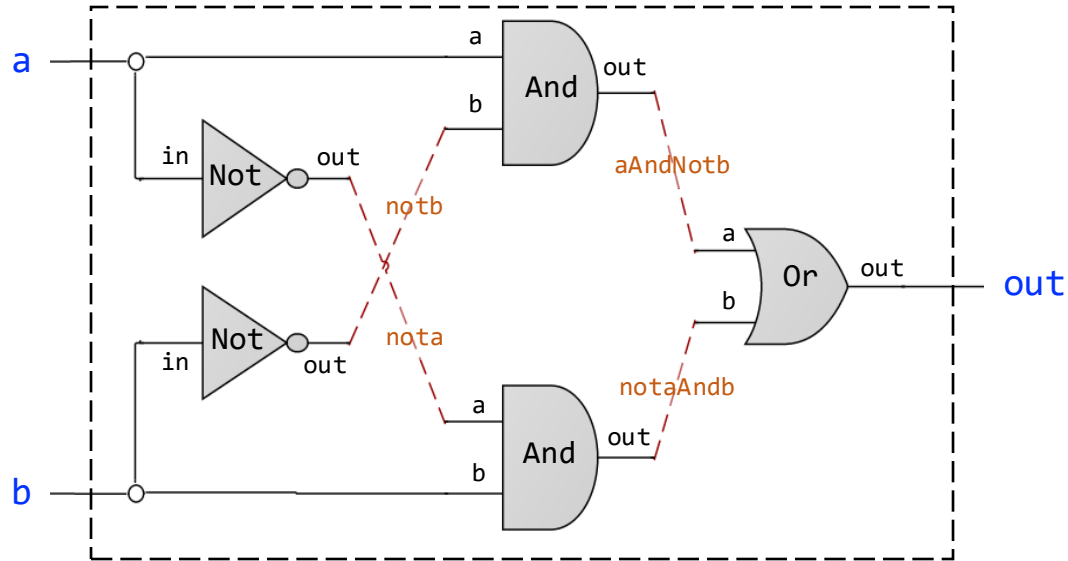
```
And (a=, b=, out= );
```

```
Or (a=, b=, out= );
```

```
Xor (a=, b=, out= );
```

```
...
```

# Design: Implementation



```
/** 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 set (APIs): */
```

```
...
```

```
Not (in=, out= );
```

```
And (a=, b=, out= );
```

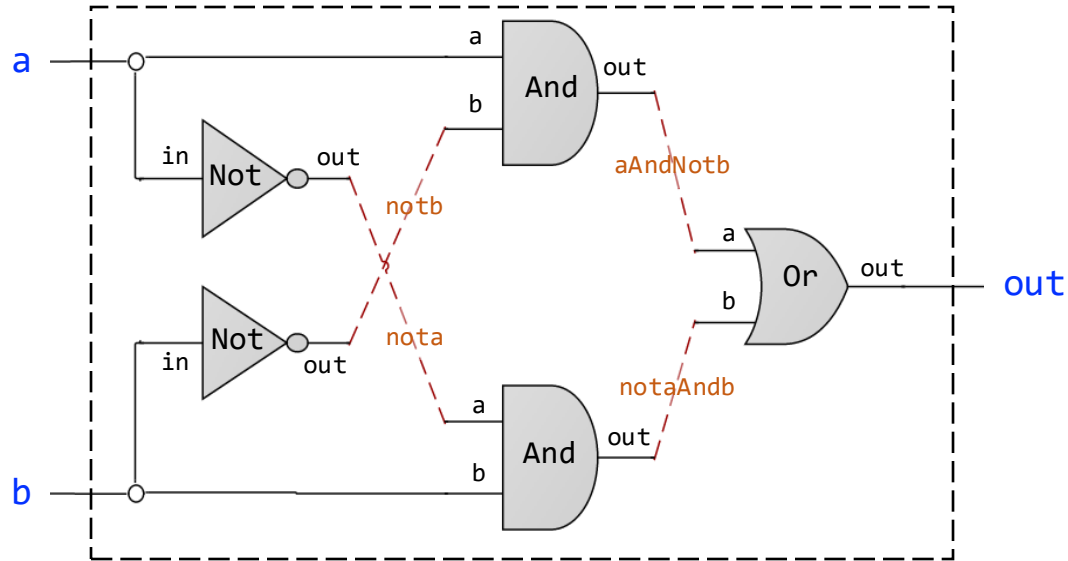
```
Or (a=, b=, out= );
```

```
Xor (a=, b=, out= );
```

```
...
```

Question: What is the missing HDL line?

# Design: Implementation

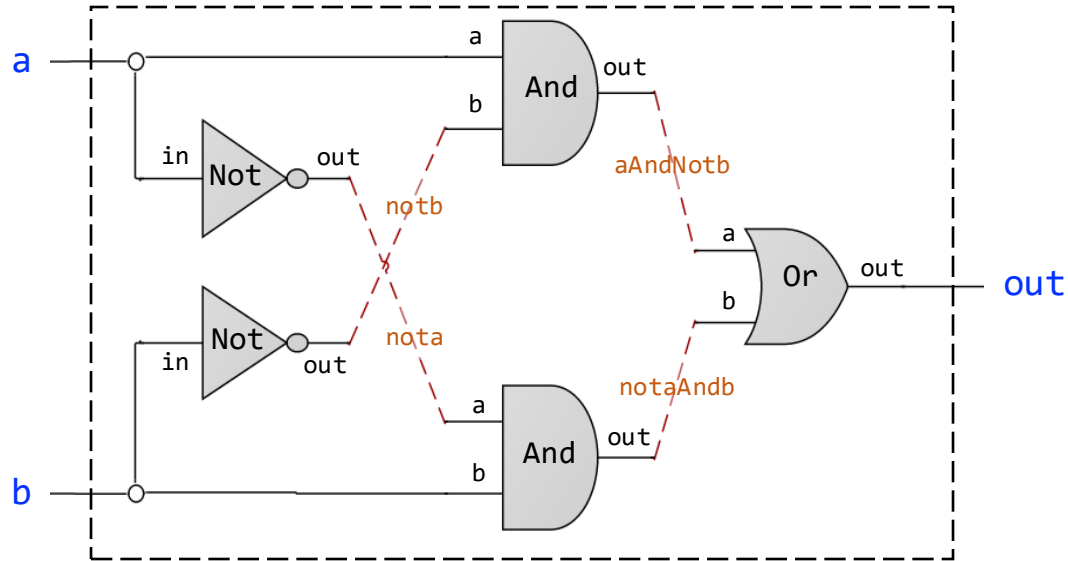


```
/** out = (a And Not(b)) Or (Not(a) And b) */  
CHIP Xor {  
  IN a, b;  
  OUT out;  
  PARTS:  
    Not (in=a, out=nota);  
    Not (in=b, out=notb);  
    And (a=a, b=notb, out=aAndNotb);  
    And (a=nota, b=b, out=notaAndb);  
    Or (a=aAndNotb, b=notaAndb, out=out);  
}
```



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

# Interface / Implementation



gate interface {

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

gate Implementation }

A logic gate has:

- One interface
- Many possible implementations

# Hardware description languages

---

## Observations

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

```
/** 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
- Documentation:  
[The Elements of Computing Systems / Appendix 2: HDL](#)

```
/** 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
- Nand

## Practice



Logic gates



HDL



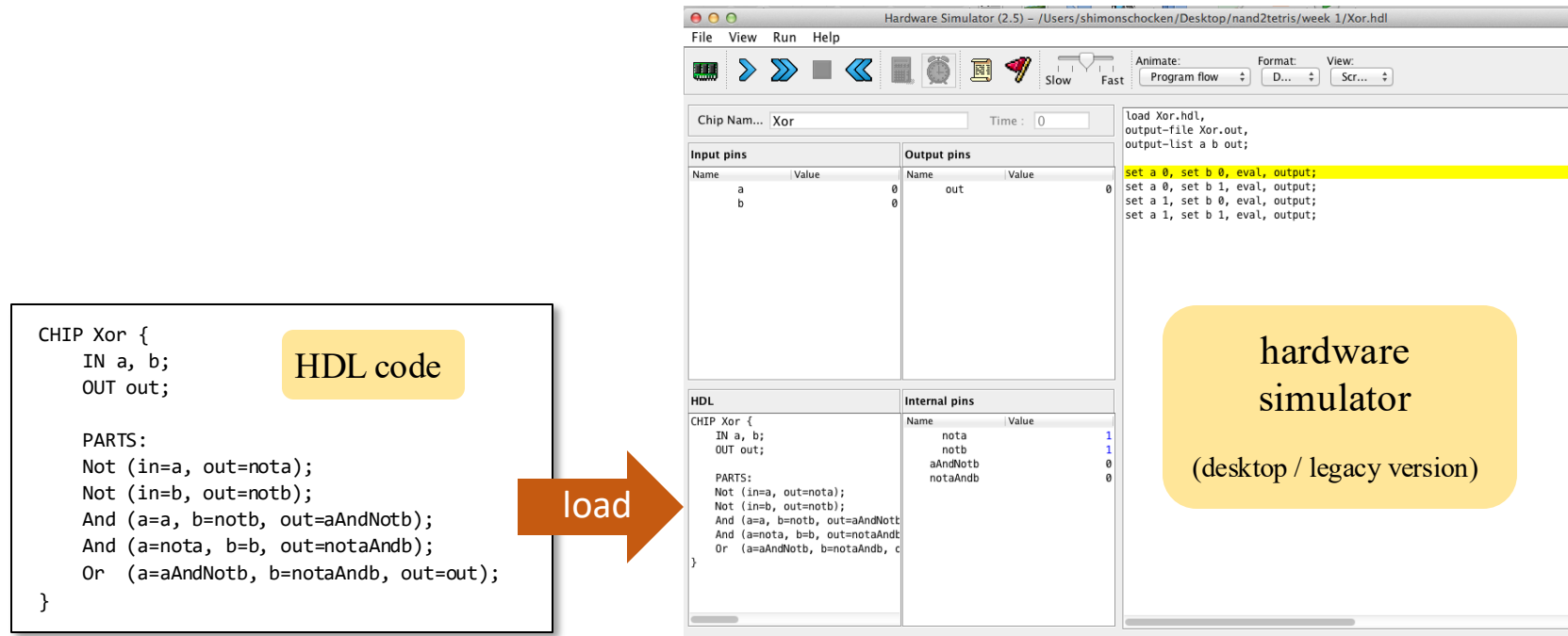
Hardware simulation

- Multi-bit buses

## Project 1

- Introduction
- Chips
- Guidelines

# Hardware simulation (in a nutshell)

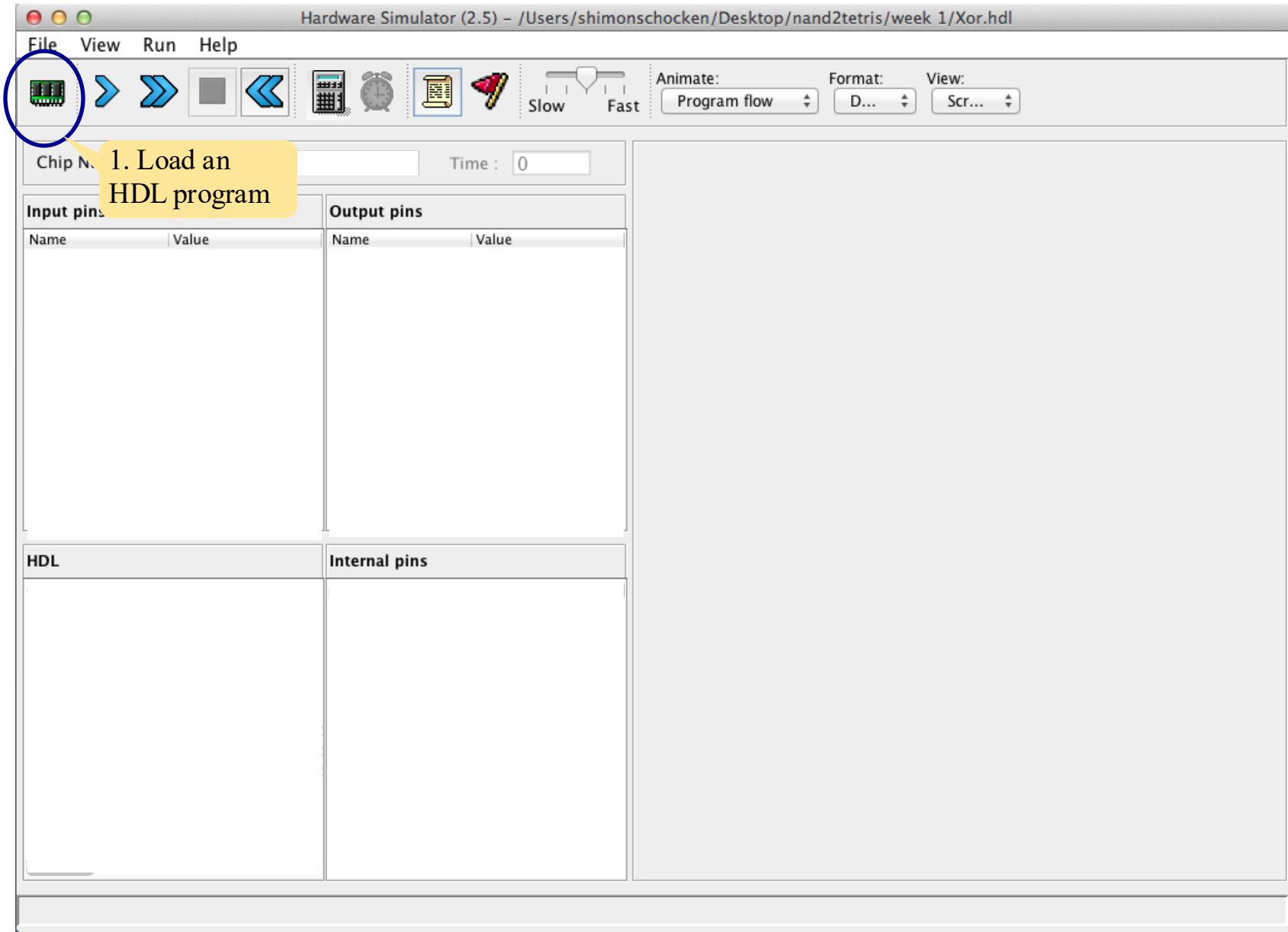


## Simulation options

➡ Interactive

- Script-based.

# Interactive simulation



# Interactive simulation

The screenshot displays the Hardware Simulator (2.5) interface. The title bar indicates the file path: `/Users/shimonschocken/Desktop/nand2tetris/week 1/Xor.hdl`. The interface includes a menu bar (File, View, Run, Help) and a toolbar with icons for loading a program, running, pausing, and evaluating logic. The main workspace is divided into several panels:

- Input pins:** A table with columns 'Name' and 'Value'. It shows inputs 'a' and 'b' with values 0 and 1 respectively.
- Output pins:** A table with columns 'Name' and 'Value'. It shows output 'out' with a value of 1.
- HDL:** A text area containing the Verilog code for an XOR gate:

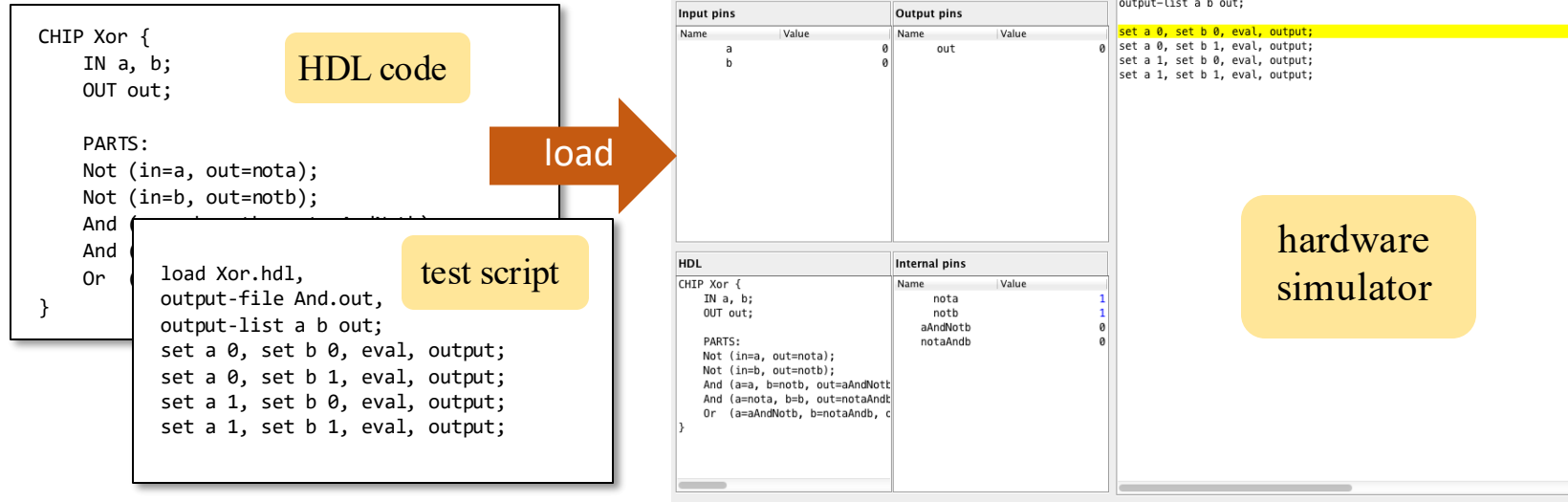
```
CHIP Xor {  
  IN a, b;  
  OUT out;  
  
  PARTS:  
    Not (in=a, out=nota);  
    Not (in=b, out=notb);  
    And (a=a, b=notb, out=aAndNotb);  
    And (a=nota, b=b, out=notaAndb);  
    Or (a=aAndNotb, b=notaAndb, out=out);  
}
```
- Internal pins:** A table with columns 'Name' and 'Value'. It shows internal signals: 'nota' (1), 'notb' (0), 'aAndNotb' (0), and 'notaAndb' (1).

Five numbered callouts illustrate the interactive simulation steps:

1. Load an HDL program (points to the load icon in the toolbar)
2. manipulate input pins (points to the input pin values)
3. evaluate the chip logic (points to the evaluate icon in the toolbar)
4. inspect output pins (points to the output pin value)
5. inspect internal pins (points to the internal pin values)

The HDL code is also labeled with a callout: "HDL code".

# Hardware simulation in a nutshell



## Simulation options



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;
```

Xor.out

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

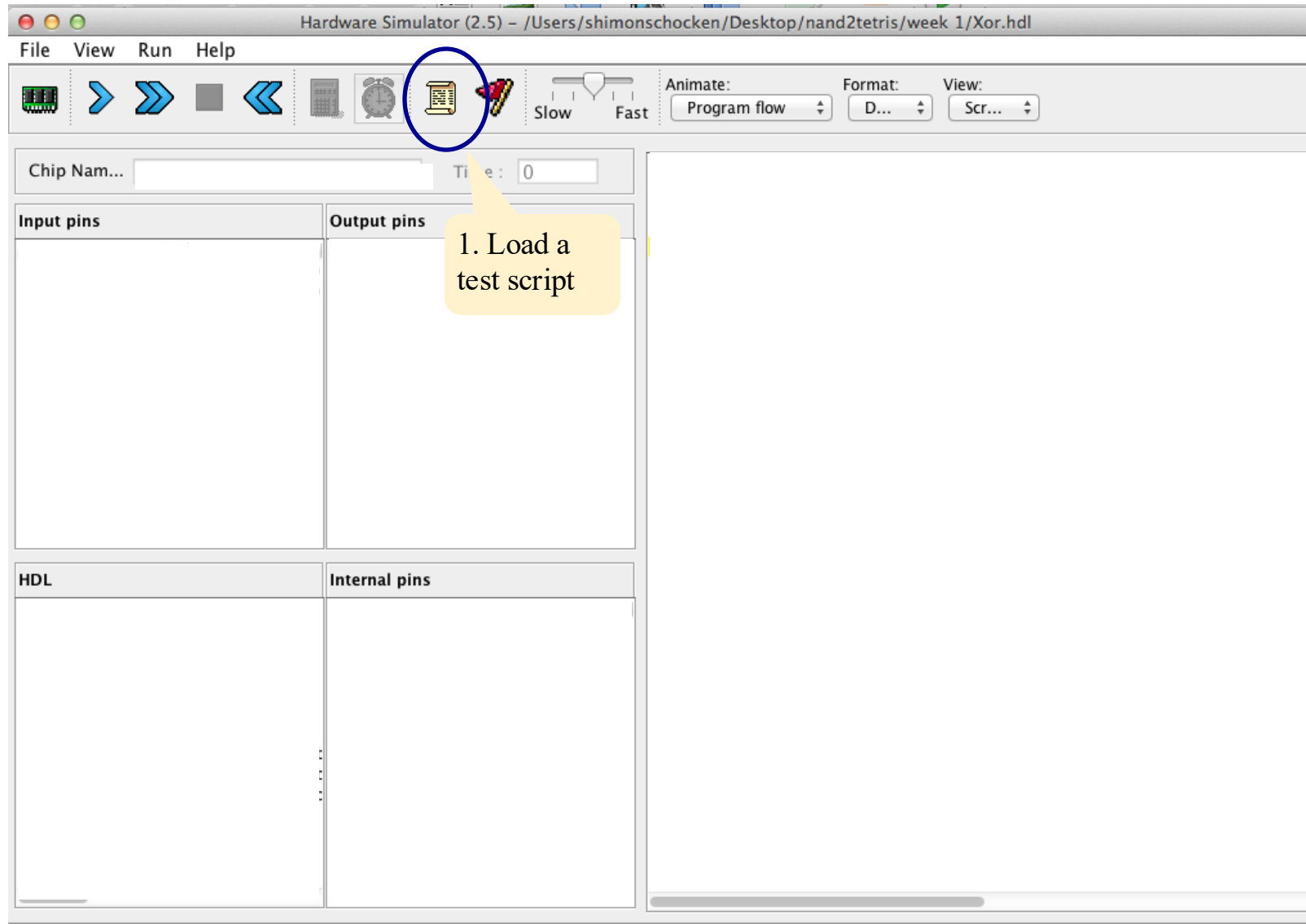
test  
script

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

## The logic of a typical test script

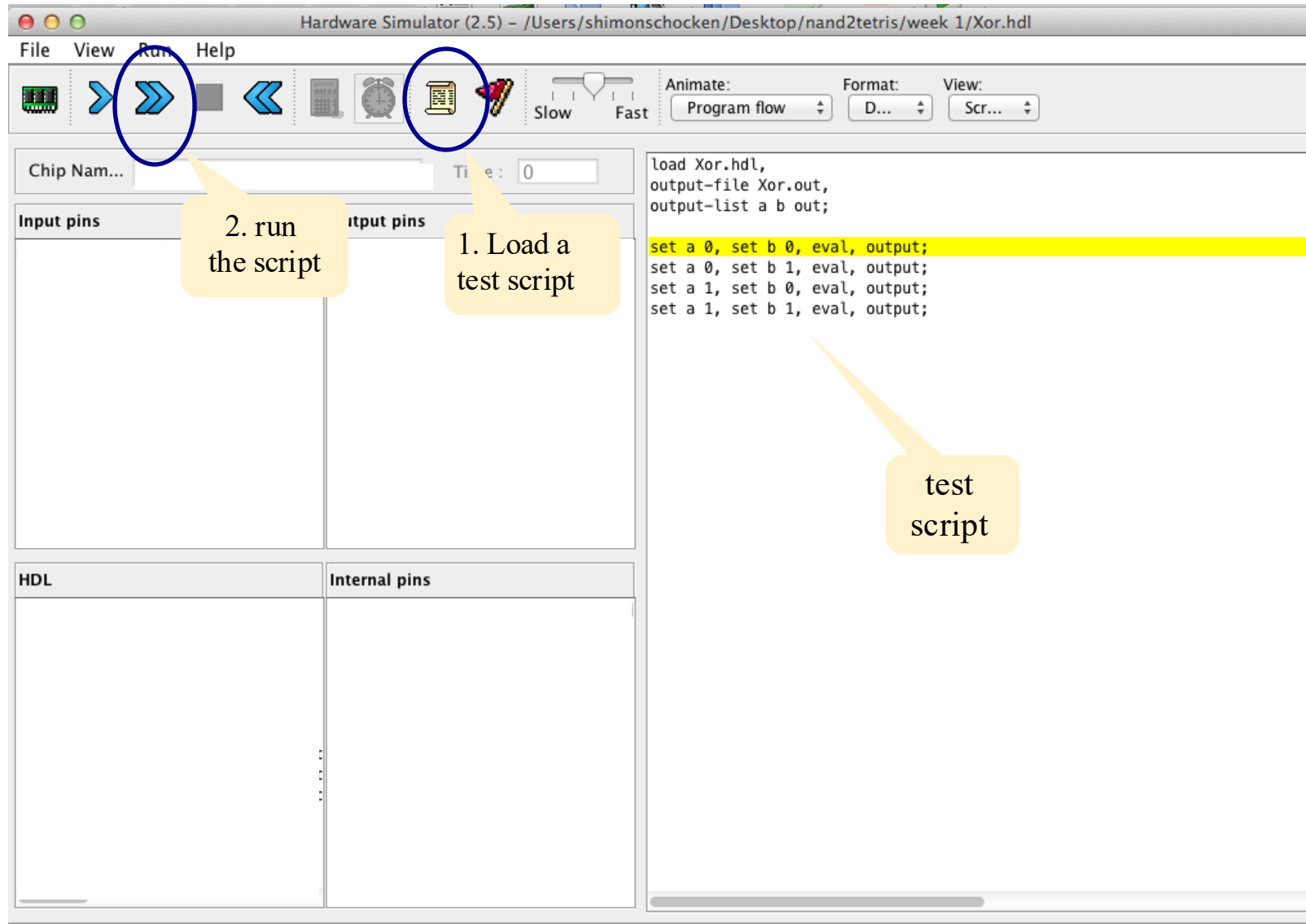
- Initialize:
  - ❑ Loads an HDL file
  - ❑ Creates an empty output file
  - ❑ Lists the names of the pins whose values will be written to the output file
- Repeat:
  - ❑ Set (inputs) , eval (chip logic) , output (print)

# Script-based simulation





# Script-based simulation



# Script-based simulation

The screenshot shows the Hardware Simulator (2.5) interface with the following components and annotations:

- Toolbar:** Contains icons for File, View, Run, and Help. The Run icon (a blue double arrow) is circled in blue and labeled "2. run the script". The Load icon (a document with a red ribbon) is circled in blue and labeled "1. Load a test script".
- Chip Name:** Set to "Xor".
- Input pins:** A table with columns "Name" and "Value". It shows inputs "a" and "b", both with a value of 0.
- Output pins:** A table with columns "Name" and "Value". It shows output "out" with a value of 0.
- HDL Code:** A text area containing the HDL code for the Xor chip. A yellow callout labeled "HDL code" points to this area.
- Internal pins:** A table with columns "Name" and "Value". It shows internal signals: "nota" (1), "notb" (1), "aAndNotb" (0), and "notaAndb" (0).
- Script Editor:** A text area containing a test script. A yellow callout labeled "test script" points to this area. The script is:

```
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;
```
- View:** A dropdown menu set to "Scr..." (Script). A yellow callout labeled "inspect the output file" points to this menu.
- Xor.out:** A table showing the output of the simulation. A yellow callout labeled "Xor.out" points to this table.

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

# 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;
```

## 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.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

compare

# 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;
```

## Demos:

[Experimenting with Built-In Chips](#)

[Building and Testing HDL-based Chips](#)

[Script-Based Chip Testing](#)

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





# Chapter 1: Boolean logic

---

## Theory

- Basic concepts
- 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 called “bus”

## Example: 16-bit bus

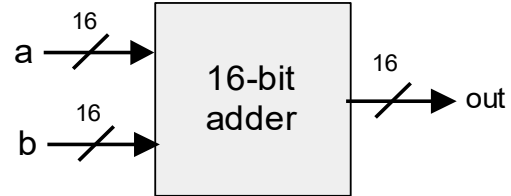
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	0	0	0	1	1	0	1	1	1	0	1

MSB = Most  
significant bit

LSB = Least  
significant bit

# Working with buses: Examples

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



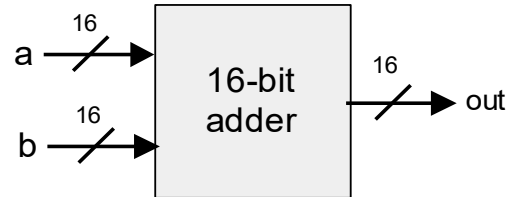
	15	...	1	0
a:	1	...	1	1
b:	0	...	1	0
c:	0	...	0	1
out:	1	...	1	0

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



# Working with buses: Examples

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

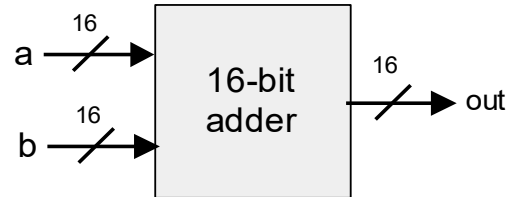


	15	...	1	0
a:	1	...	1	1
b:	0	...	1	0
c:	0	...	0	1
out:	1	...	1	0

```
/* 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: Examples

```
/* 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;  
  ...  
}
```

a: 

3	2	1	0
0	1	1	1

out: 

0
---

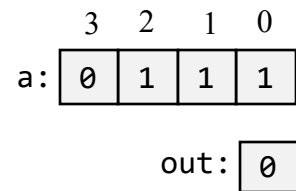
```
/* 4-way And: Ands 4 bits. */
```

```
CHIP And4Way {  
  IN a[4];  
  OUT out;
```

# 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=      );
```

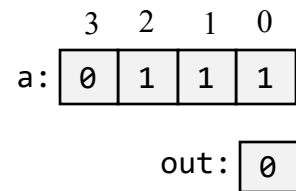
```
    And(a=      , b=      , out=      );
```

```
}
```

# Working with individual bits within buses

```
/* Returns 1 if a==1 and b==1,  
   0 otherwise. */
```

```
CHIP And {  
  IN a, b;  
  OUT out;  
  ...  
}
```



```
/* 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
---

3 2 1 0  
a: 

0	1	0	1
---	---	---	---

  
b: 

0	0	1	1
---	---	---	---

  
out: 

0	0	0	1
---	---	---	---

```
/* 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.

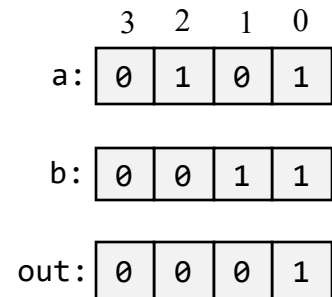
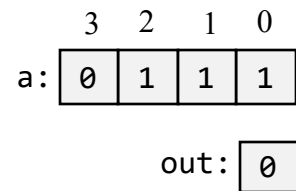
```
/* Bit-wise And of two 4-bit inputs */
```

```
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;  
  ...  
}
```



```
/* 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.

```
/* 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
---

3 2 1 0  
a: 

0	1	0	1
---	---	---	---

b: 

0	0	1	1
---	---	---	---

out: 

0	0	0	1
---	---	---	---

```
/* 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.

```
/* 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

Additional essential tips:

[\*The Elements of Computing Systems / Appendix 2: HDL\*](#)



# Chapter 1: Boolean logic

---



## Theory

- Basic concepts
- Nand



## Practice

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

## Project 1

- Introduction
- Chips
- Guidelines

# Chapter 1: Boolean logic

---

## Theory

- Basic concepts
- 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);
}
```

HDL implementation

Xor.hdl

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

    BUILTIN Xor;
    // Implemented in Java / Javascript / ...
}
```

Internal implementation

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 hardware simulator](#)

# Hardware construction projects

---

## Key players:

### Architect

- ❑ Decides which chips are needed
- ❑ Specifies the chips

### Developers

- ❑ Build / test the chips



## In Nand to Tetris:

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

For each chip, the architect supplies:

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

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

These files specify:

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

compare  
file

Xor.cmp

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

# 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

These files specify:

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

compare  
file

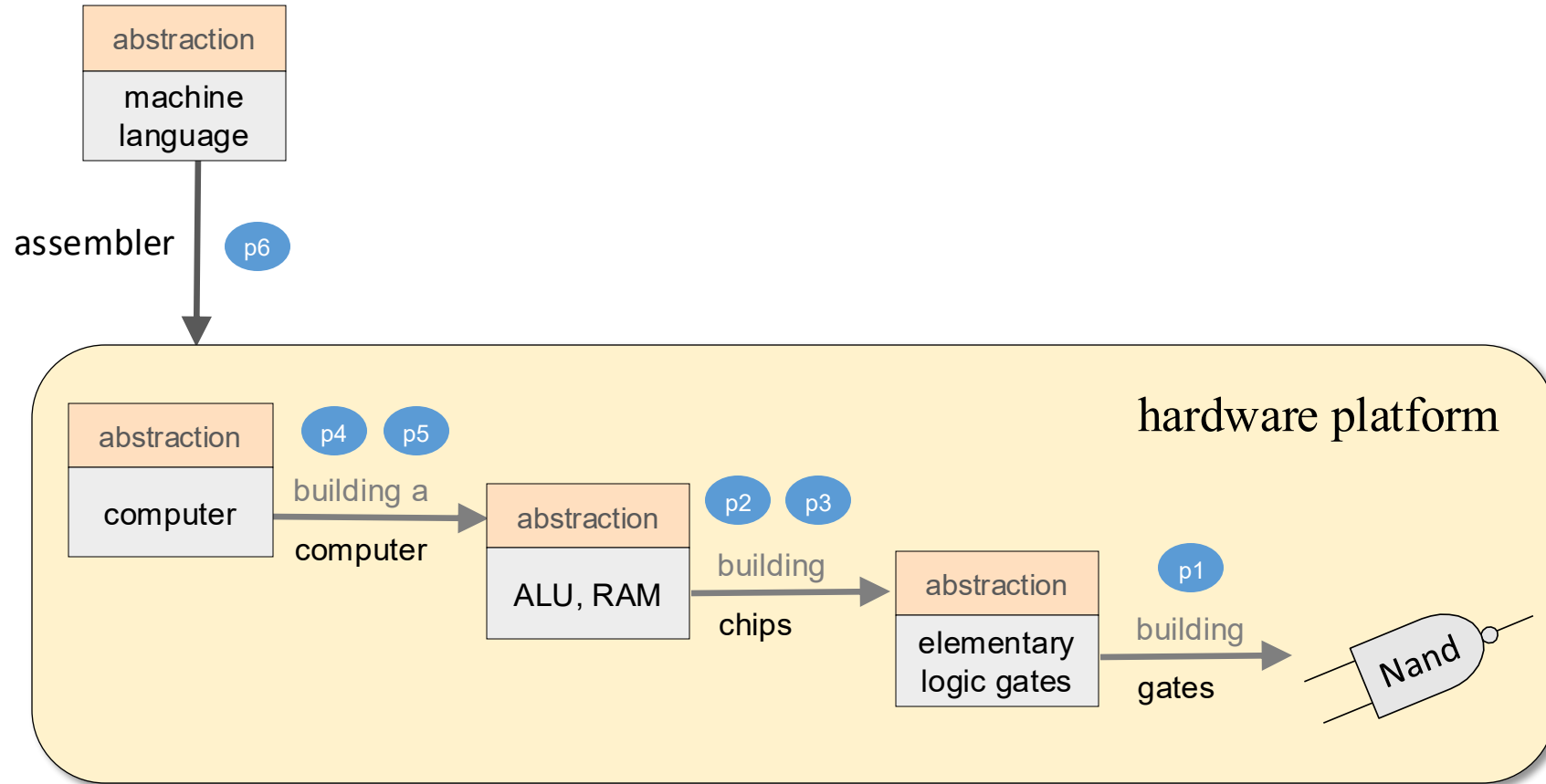
Xor.cmp

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

The developer's task:

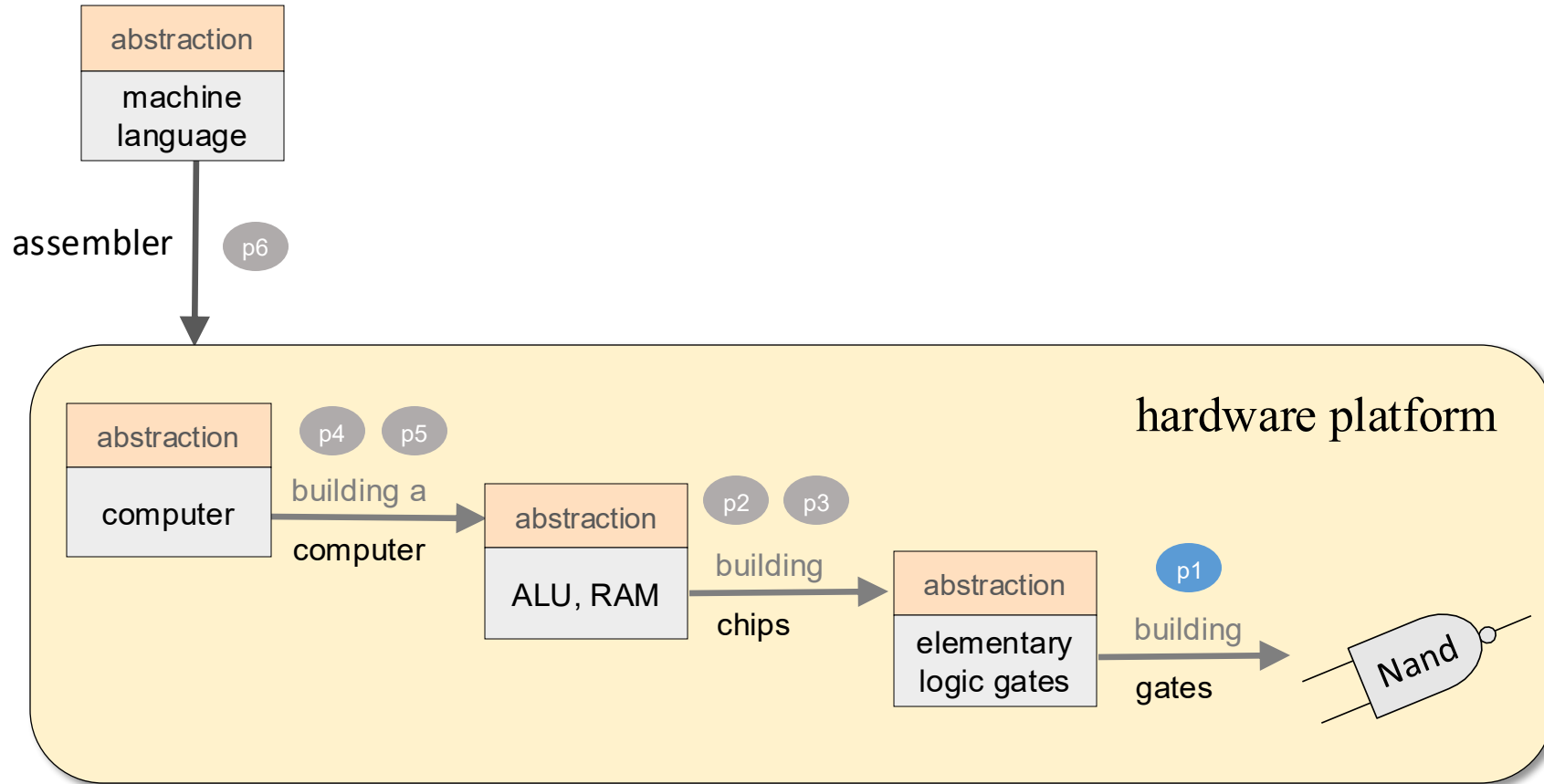
Implement the chip (complete the supplied .hdl file),  
using these resources.

# Project 1



# Project 1

---



## Project 1

Build 15 elementary logic gates



# 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

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

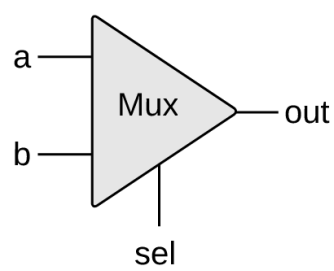
- Not16
- And16
- Or16
- Mux16

## Multi-way variants

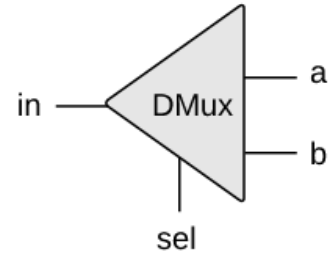
- Or8Way
- Mux4Way16
- Mux8Way16
- DMux4Way
- DMux8Way

# Multiplexor / Demultiplexor

---



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

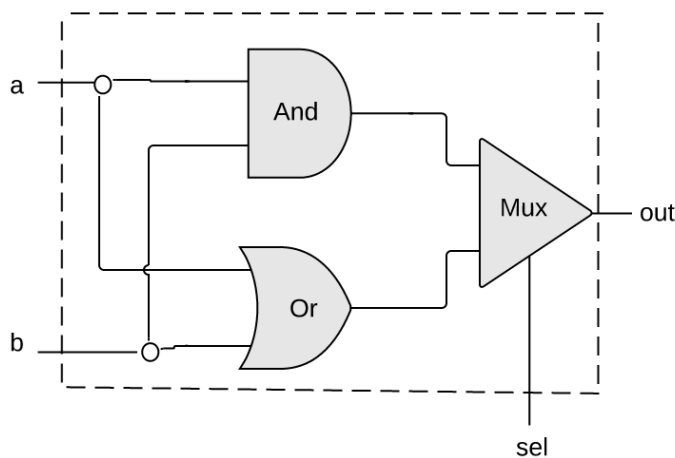
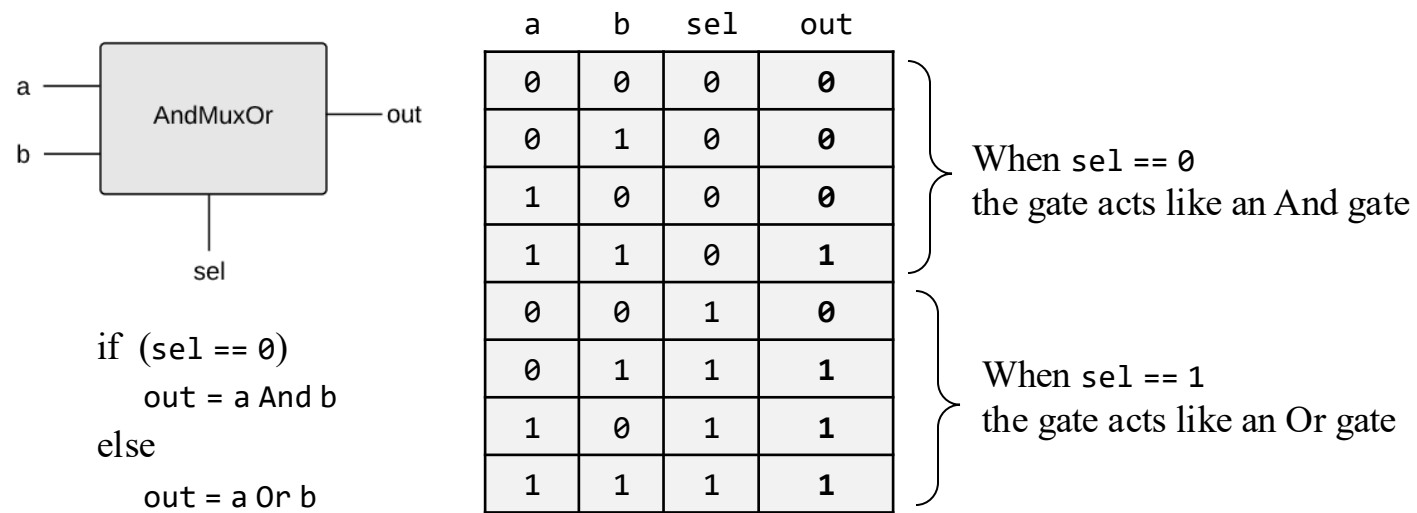


```
if (sel == 0)
  {a, b} = {in, 0}
else
  {a, b} = {0, in}
```

## Widely used in:

- Hardware design
- Communications networks.

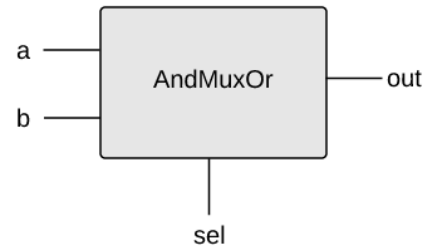
# Example 1: Using Mux logic to build a programmable gate



Mux.hdl

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

# Example 1: Using Mux logic to build a programmable gate

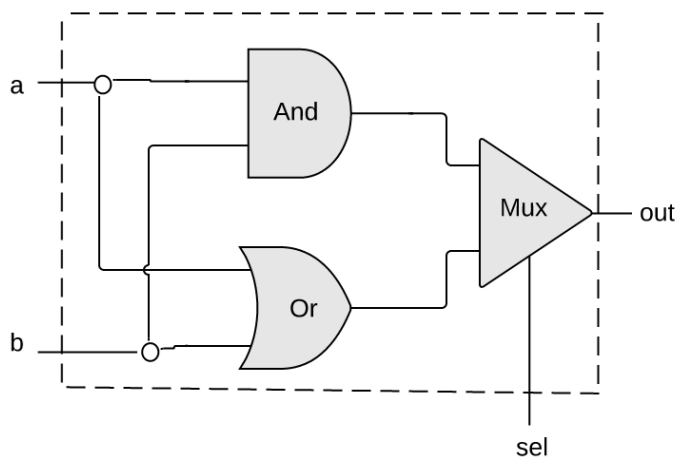


if (sel == 0)  
    out = a And b  
else  
    out = a Or b

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

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

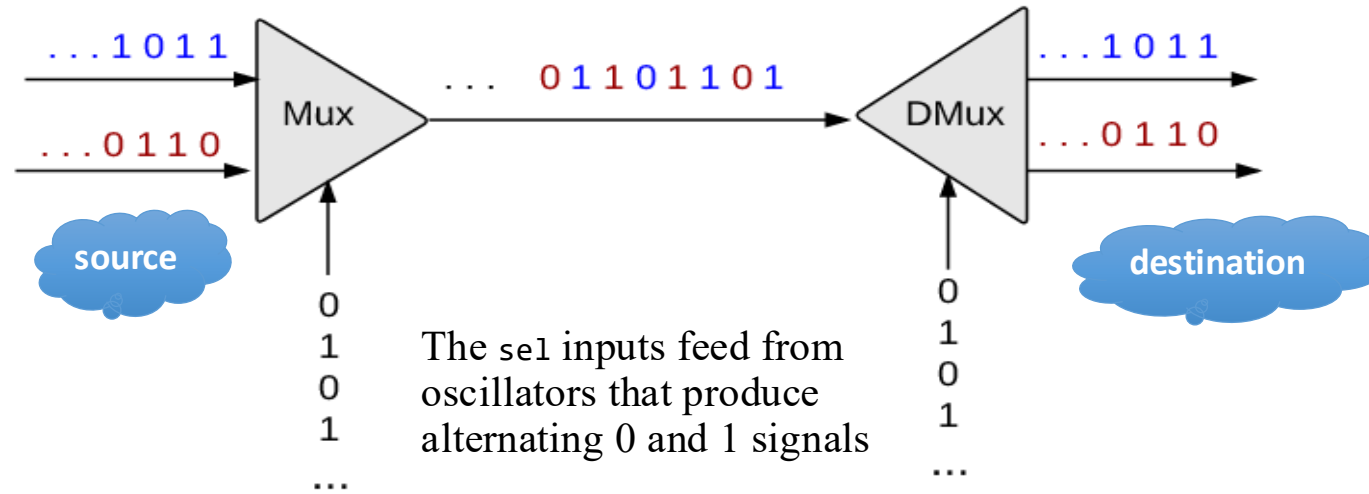
When sel == 1  
the gate acts like an Or gate



Mux.hdl

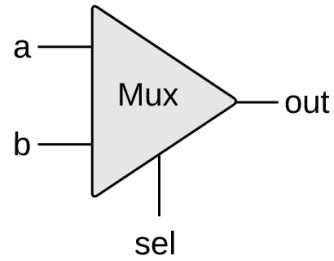
```
CHIP AndMuxOr {  
    IN a, b, sel;  
    OUT out;  
  
    PARTS:  
    And (a=a, b=b, out=andOut);  
    Or (a=a, b=b, out=orOut);  
    Mux (a=andOut, b=orOut, sel=sel, out=out);  
}
```

## 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

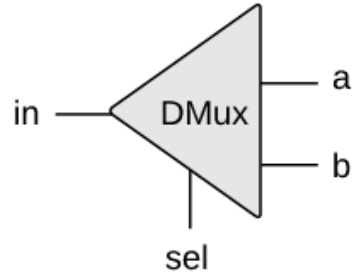
```
CHIP Mux {
    IN a, b, sel;
    OUT out;

    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

- 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:
}
```

## Implementation tip

Simple truth table, simple 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


# Project 1

---

## Elementary logic gates

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

## 16-bit variants

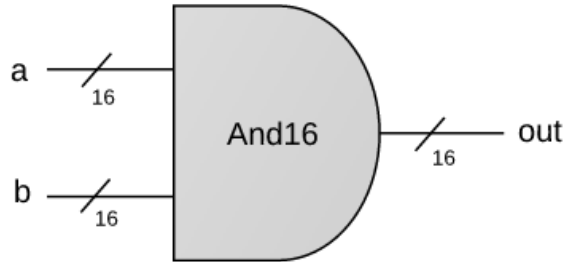
- Not16
-  □ And16
- Or16
- Mux16

## Multi-way variants

- Or8Way
- Mux4Way16
- Mux8Way16
- DMux4Way
- DMux8Way

# And16

---



Example:

a	=	1	0	1	0	1	0	1	1	0	1	0	1	1	0	0	
b	=	0	0	1	0	1	1	0	1	0	0	1	0	1	0	1	0
<hr/>																	
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:  
}
```

## Implementation tip

A straightforward 16-bit extension  
of the elementary And gate

(See the [HDL documentation](#)  
about 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

# Project 1

---

## Elementary logic gates

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

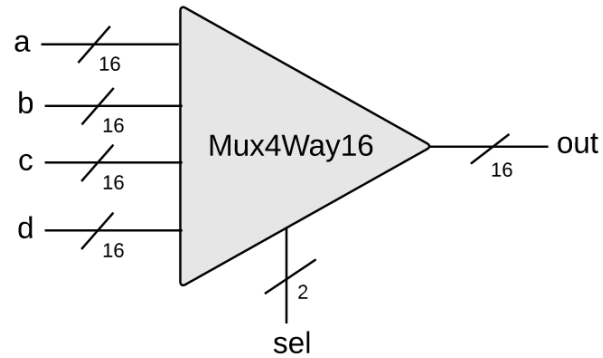
## 16-bit variants

- Not16
- And16
- Or16
- Mux16

## Multi-way variants

- Or8Way
- Mux4Way16
- Mux8Way16
- DMux4Way
- DMux8Way

# 16-bit, 4-way multiplexor



sel[1]	sel[0]	out
0	0	<b>a</b>
0	1	<b>b</b>
1	0	<b>c</b>
1	1	<b>d</b>

## 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 Mux16 gates.

# Chapter 1: Boolean logic

---

## Theory

- Basic concepts
- 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

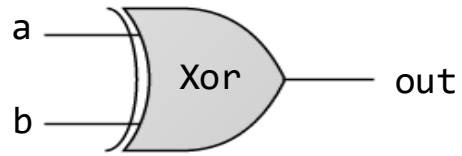


How to actually build these gates?



# 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

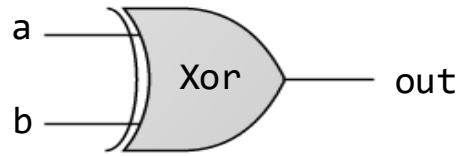
Xor.hdl (stub file)

```
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;
```

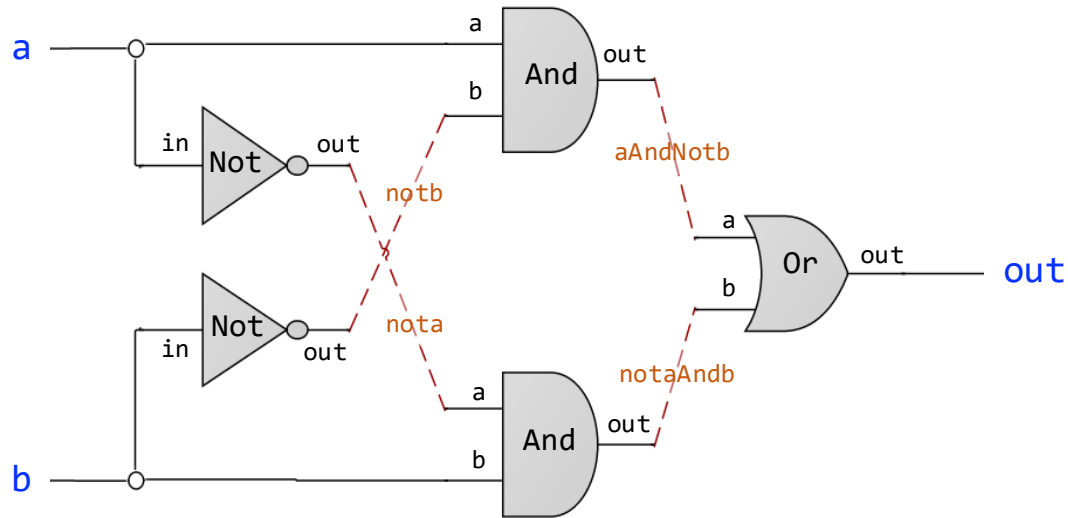
## Project 1 folder

(.hdl, .tst, .cmp files):  
nand2tetris/projects/1

## 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= ,address= );
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= ,out= );
```

```
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-part, say `Bar`:

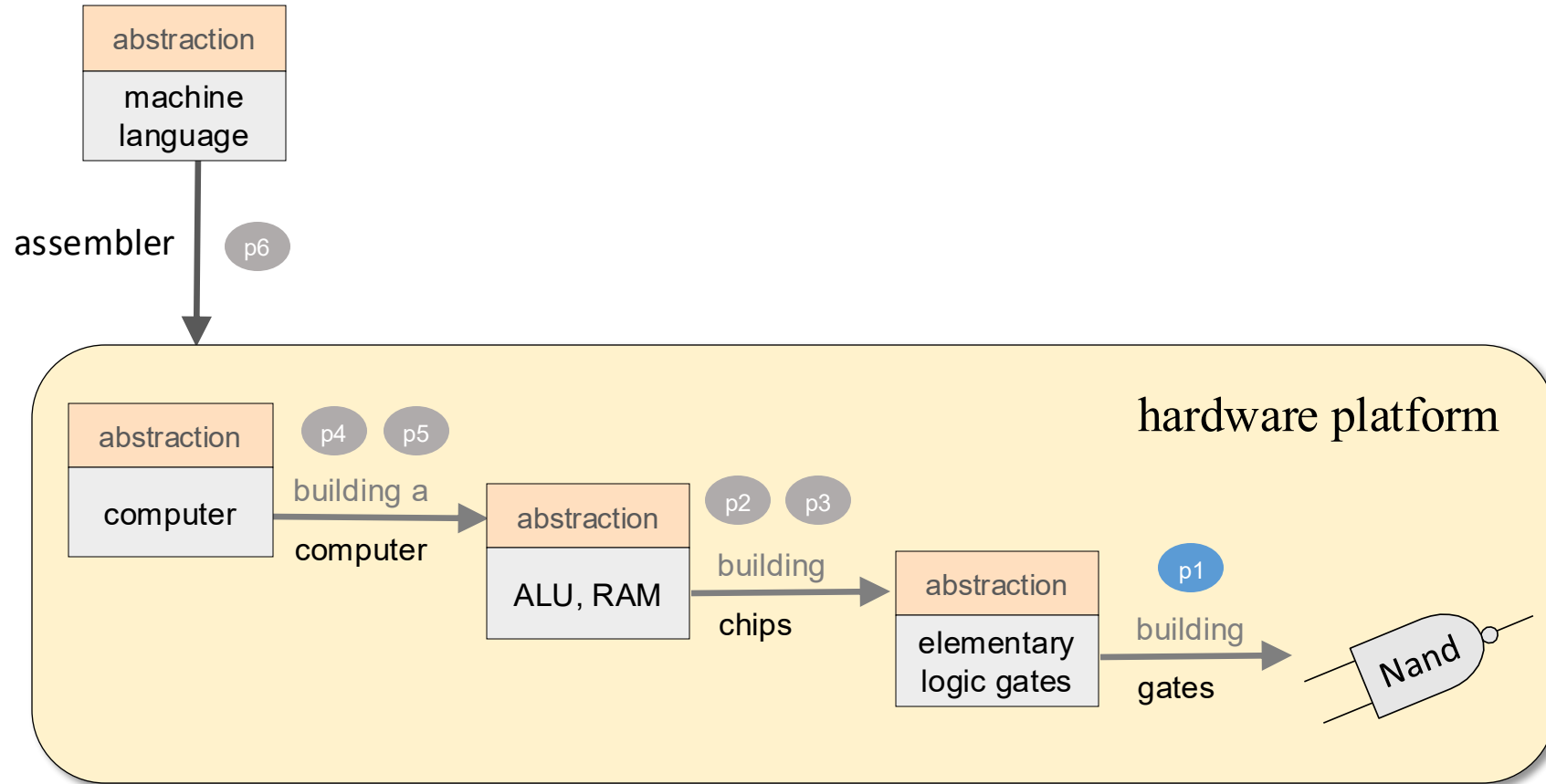
## If using the web-based IDE:

When the chip-part has no HDL implementation, the simulator uses the builtin version

## If using the desktop hardware simulator:

- Typically, `Bar.hdl` is a stub-file, or a file that has an incomplete implementation
- If you want to use `Bar` as a built-in chip:  
Remove the file `Bar.hdl` from the project folder (or rename it, say, `Bar1.hdl`)
- The result: 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.

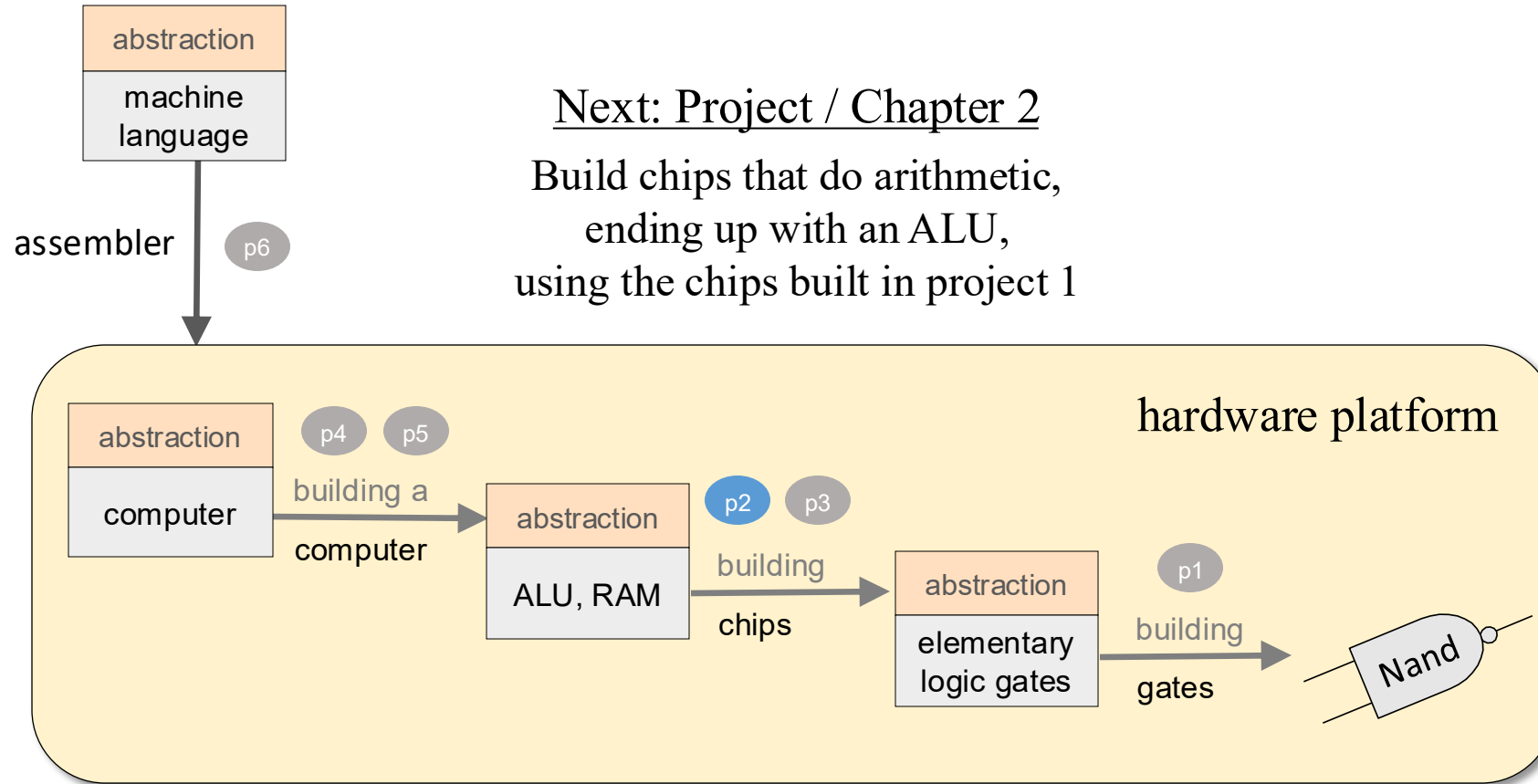
# What's next?



## Project / Chapter 1

Build 15 elementary logic gates

# What's next?



## Next: Project / Chapter 2

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

## Project / Chapter 1

Build 15 elementary logic gates