Lecture 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
- 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`

Different labels, all referring to two possible states.

George Boole

1815 - 1864

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

…

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

Boolean values

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

1 binary variable: 2 possible states

2 binary variables: 4 possible states

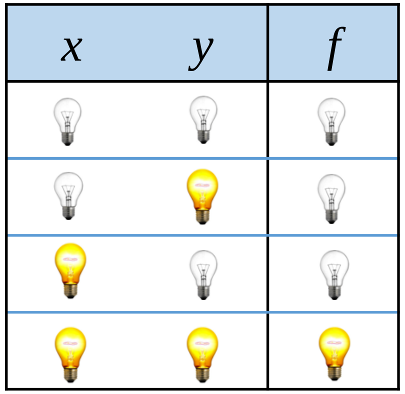
3 binary variables: 8 possible states

. . .

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

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

Boolean functions

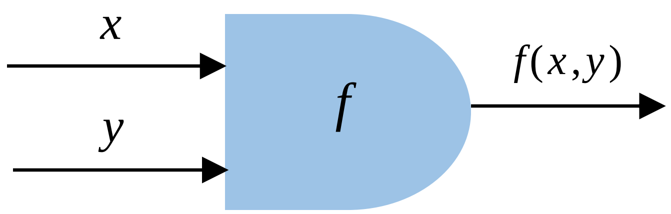


Boolean functions

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

Boolean functions

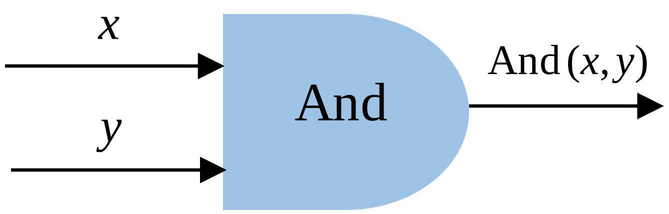
x	y	f
0	0	0
0	1	0
1	0	0
1	1	1



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

Boolean functions

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



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

<u>Boolean function</u>

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 $x\ f\ y$

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

<u>Boolean function evaluation</u> (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 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 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.

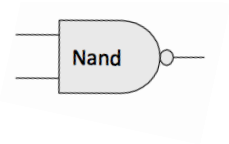
The expressive power of Nand

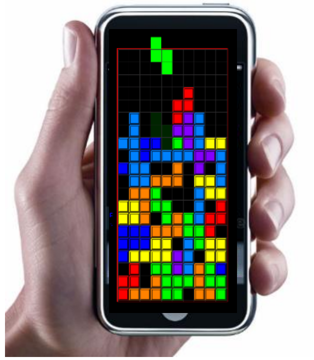
<u>Theorem:</u> Any Boolean function can be realized using only Nand.

<u>Implication:</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
- 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

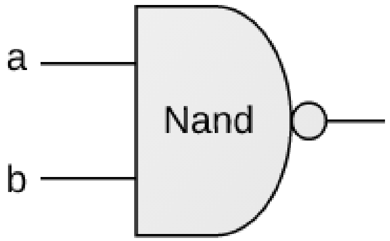
Project 1

- Introduction
- Chips
- Guidelines

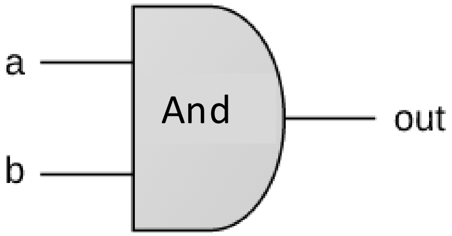
Logic gates

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

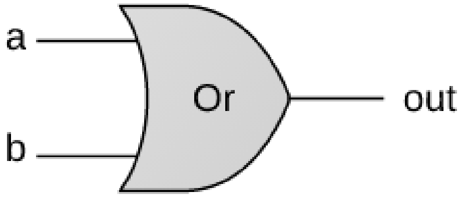
Elementary gates



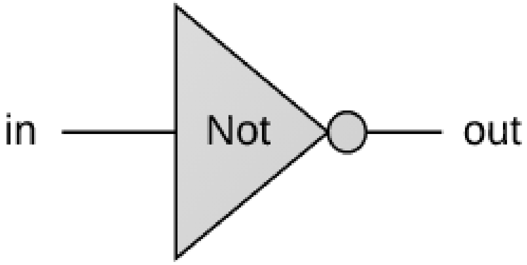

```
if (a==1 and b==1)
then out=0 else out=1
```




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




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

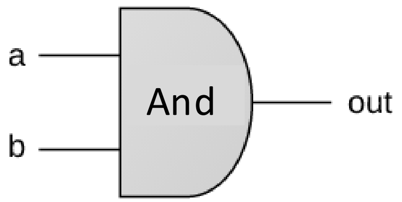



```
if (in==0)
then out=1 else out=0
```

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

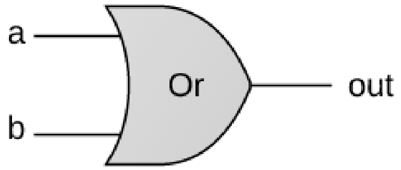
- Because either {`Nand`} or {`And`, `Or`, `Not`} (as well as other subsets of Boolean operators) can be used to span any given Boolean function
- Because they have efficient and direct hardware implementations.

Elementary gates



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

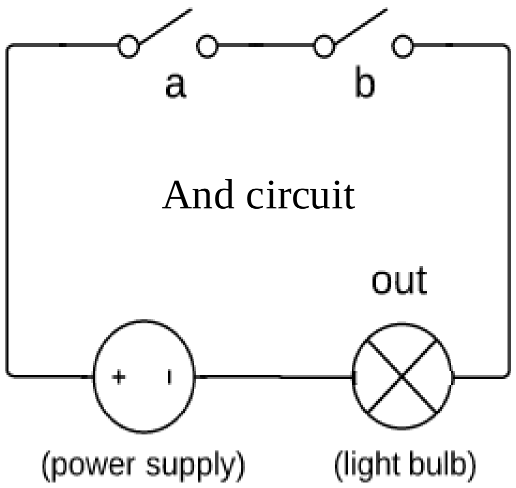
```
 if (a==1 and b==1)
then out=1 else out=0
```

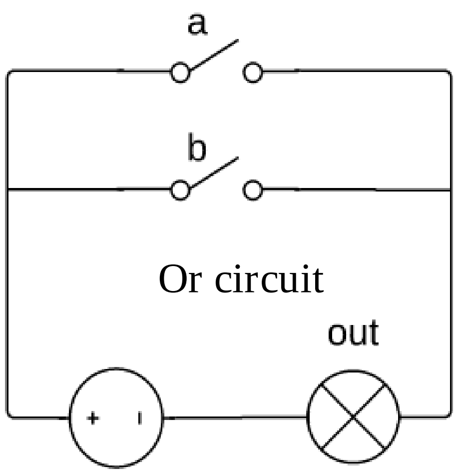


x	y	Or
0	0	0
0	1	1
1	0	1
1	1	1

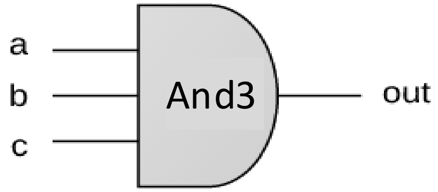
```
 if (a==1 or b==1)
then out=1 else out=0
```

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





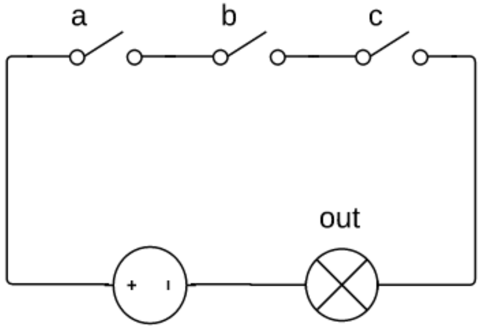
Composite gates



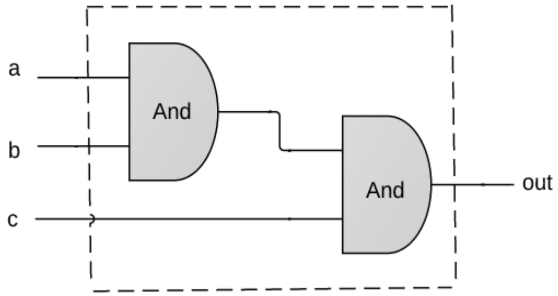

```
if (a==1 and b==1 and c==1)
   then out=1 else out=0
```

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
- Nand

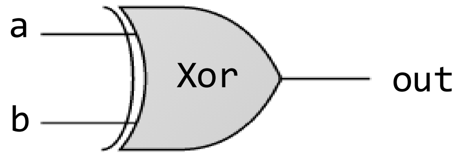
Practice

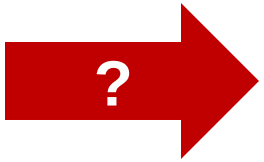
- Logic gates
- HDL
- Hardware simulation
- Multi-bit buses

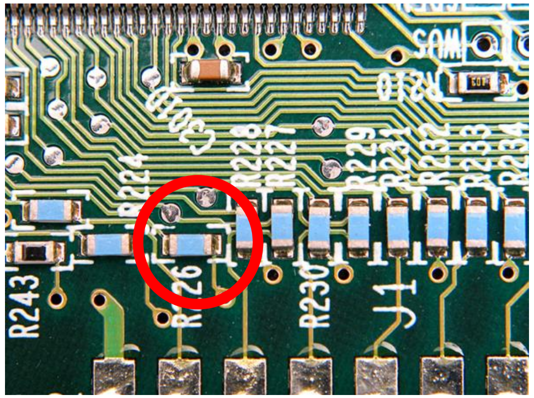
Project 1

- Introduction
- Chips
- Guidelines

Building a chip







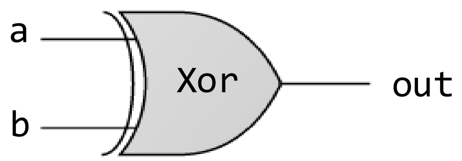
(an arbitrary image found on the Internet...)

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



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

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

<u>Requirement</u>

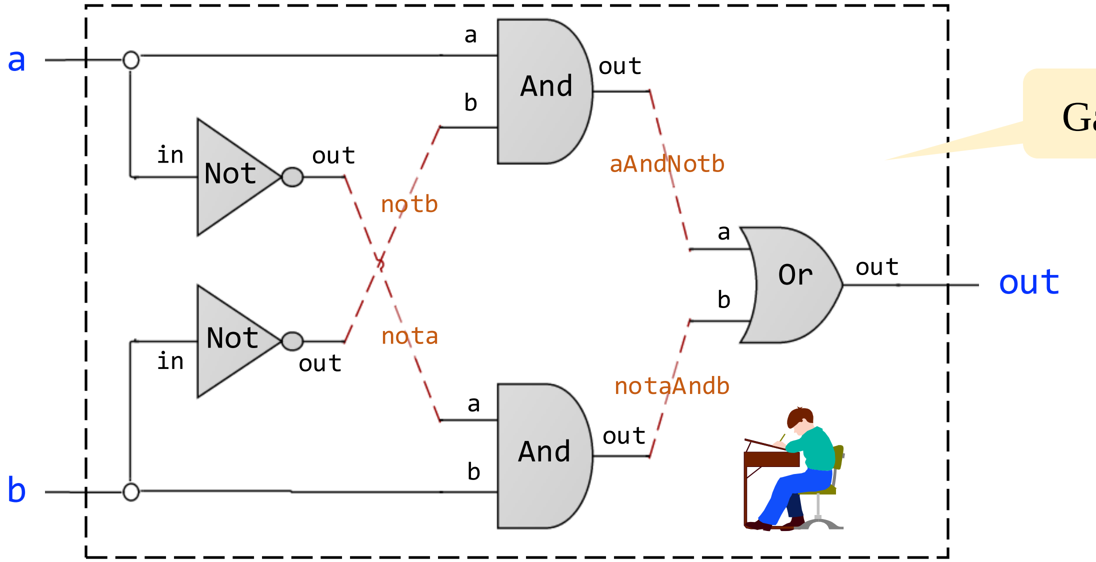
Build a chip that delivers this functionality

```
/** 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



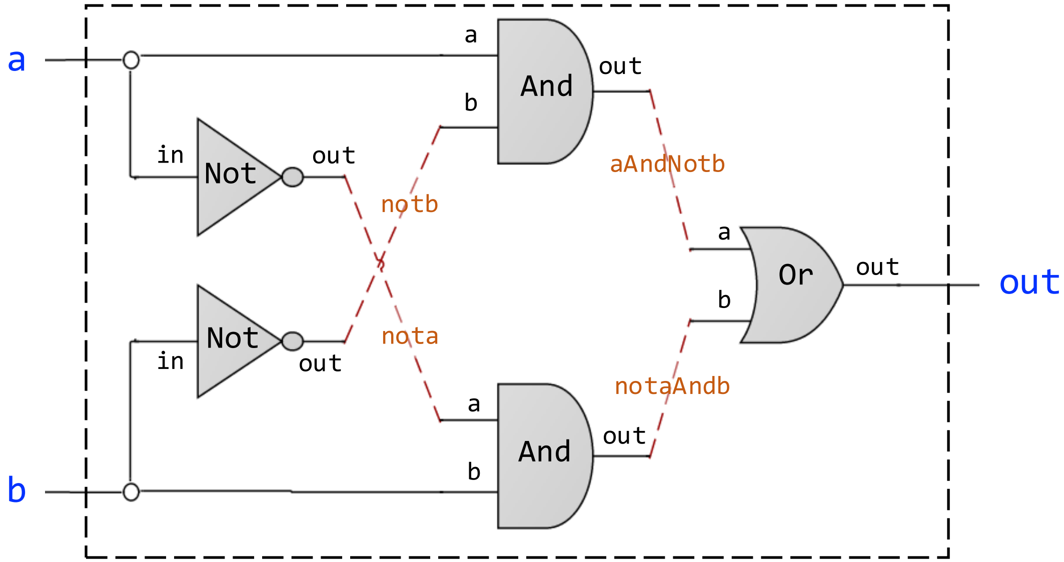
Gate diagram

```
/** 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= );
...
```

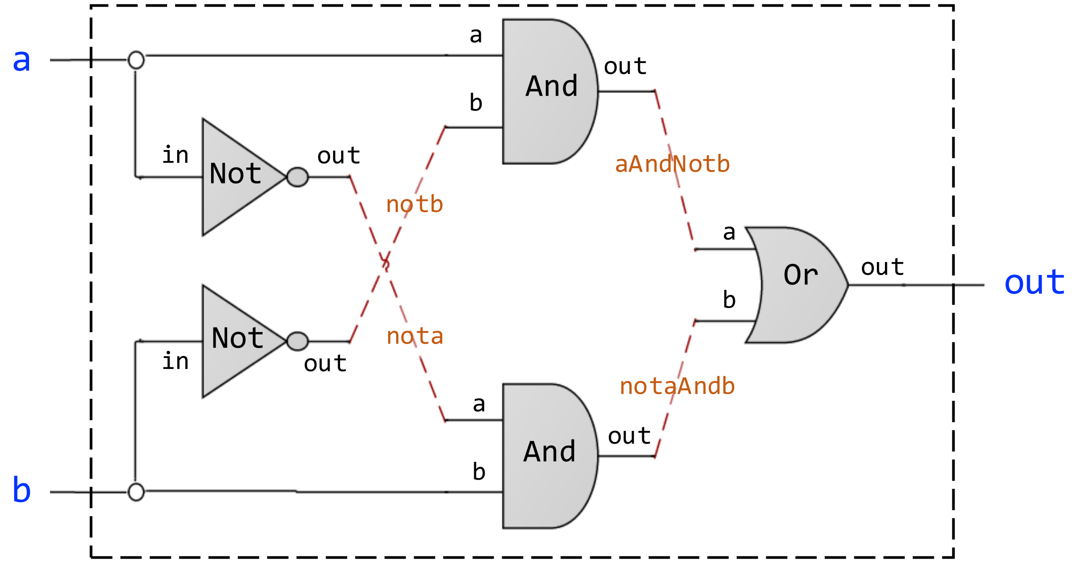
Design: Implementation




```
/** 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

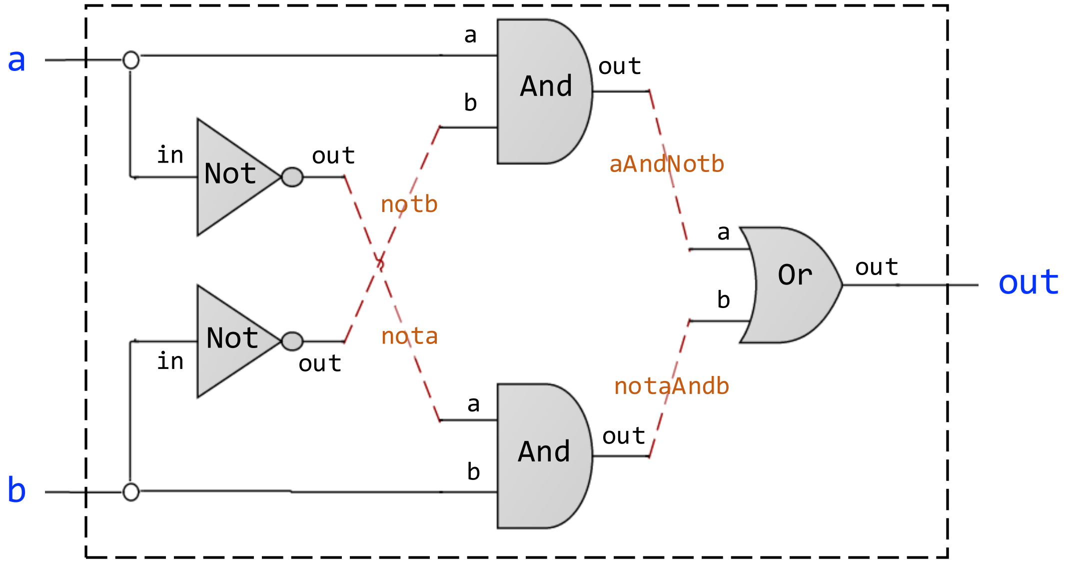



```
/** out = (a And Not(b)) Or (Not(a) And b))  */
CHIP Xor {
    IN a, b;
    OUT out;
    PARTS:
    Not (in=a, out=nota);
    Not (in=b, out=notb);
    And (a=a, b=notb, out=aAndNotb);
    And (a=nota, b=b, out=notaAndb);

}
```

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

<u>Question:</u> What is the missing HDL line?

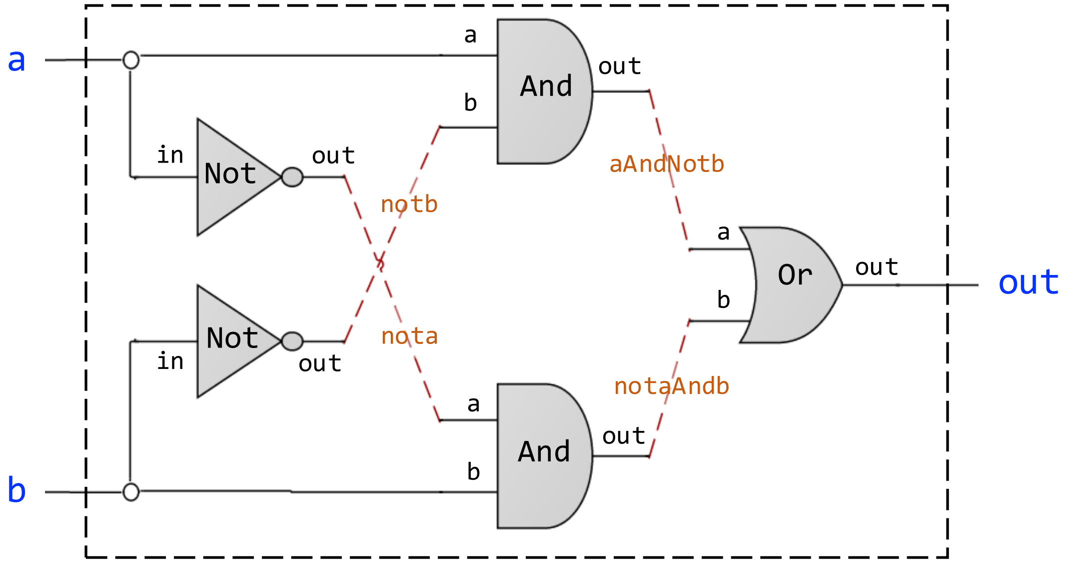
Design: Implementation




```
/** 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

gate Implement-ation

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

A logic gate has:

- One interface
- Many possible implementations

Hardware description languages

Observations

函数式 / 声明式语言　　程序员定义了他们想要实现的目标（“做什么”），而不是具体步骤（“怎么做”）。vs 命令式语言

- 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

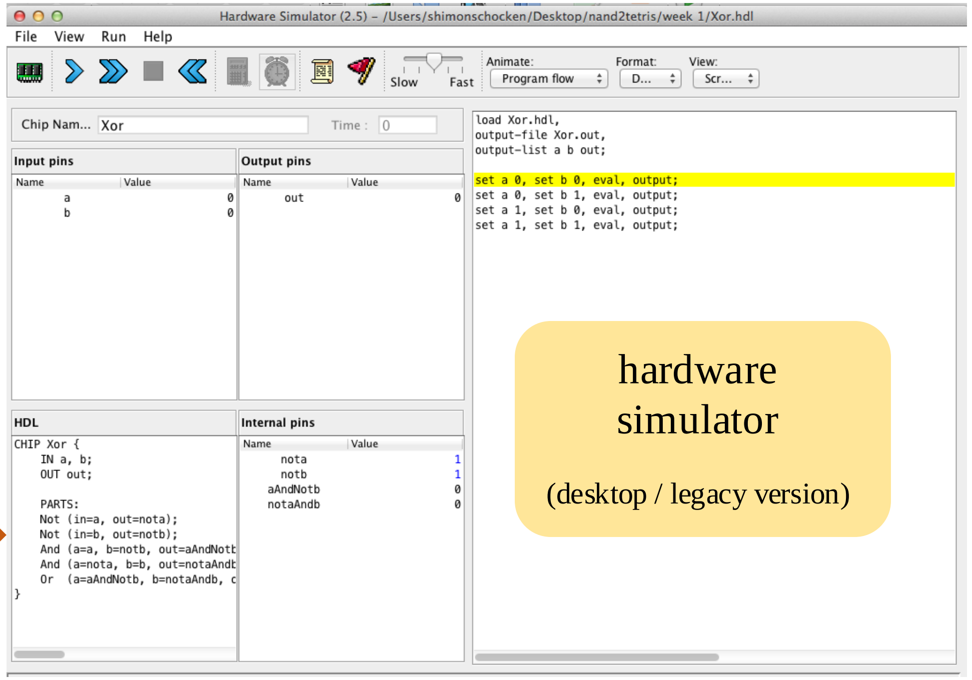
- Introduction
- Chips
- Guidelines

Hardware simulation (in a nutshell)

HDL code

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

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

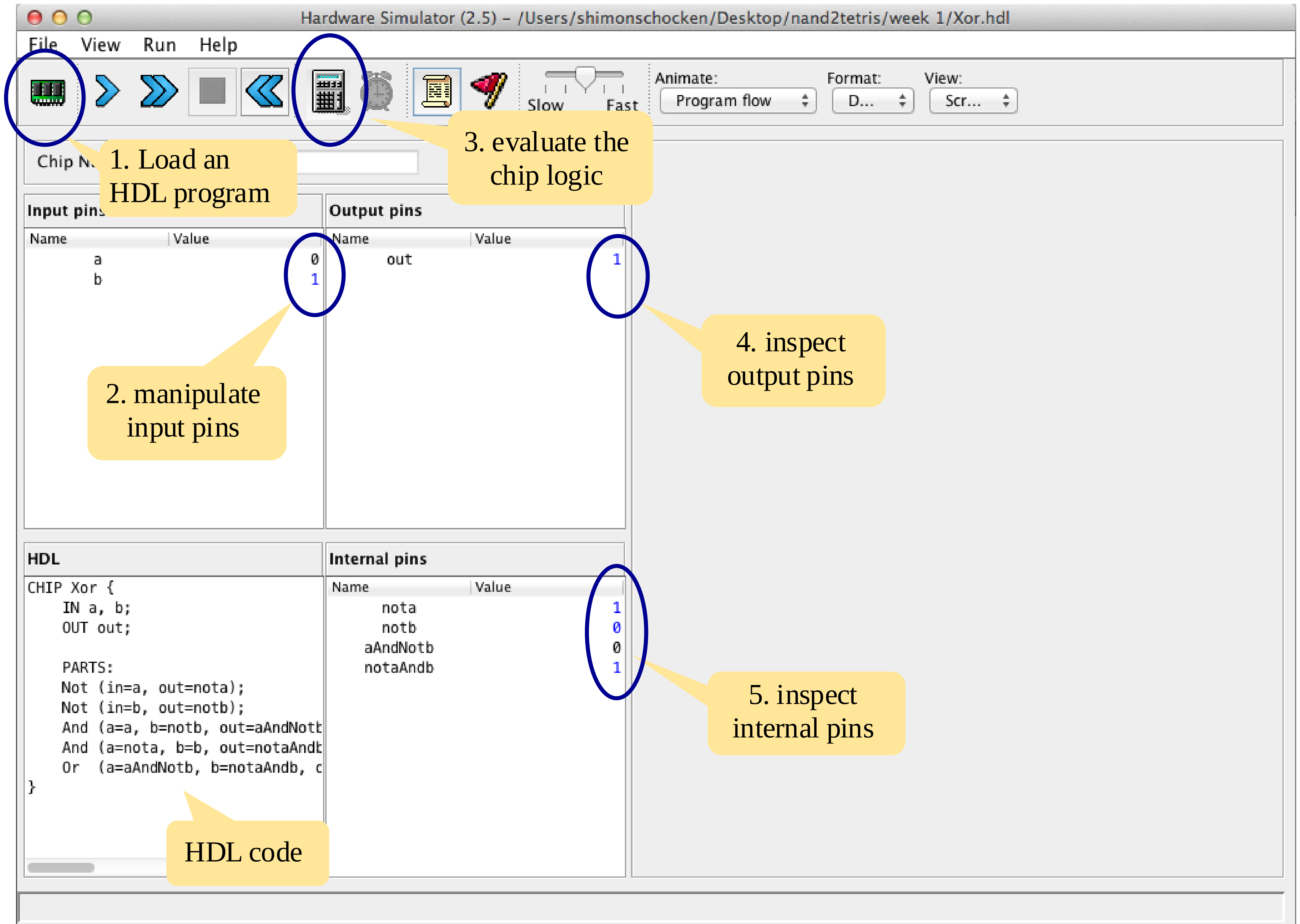
load



Simulation options

- Interactive
- Script-based.

Interactive simulation



Hardware simulation in a nutshell

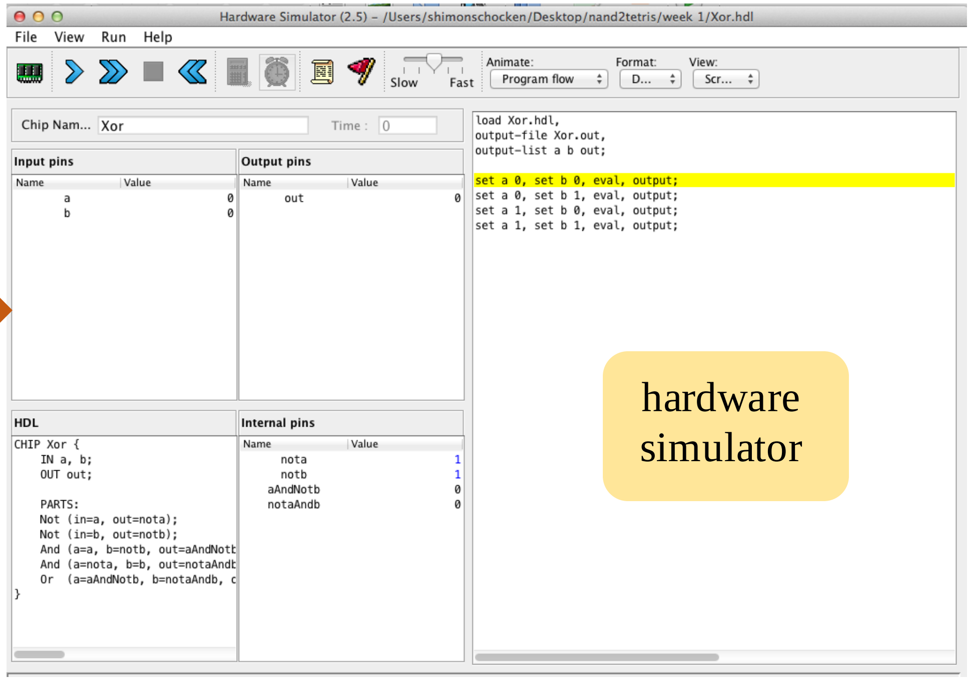
```
CHIP Xor {
    IN a, b;
    OUT out;

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

HDL code

load

```
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;
```

test script



hardware simulator

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;
```

test script

Xor.out

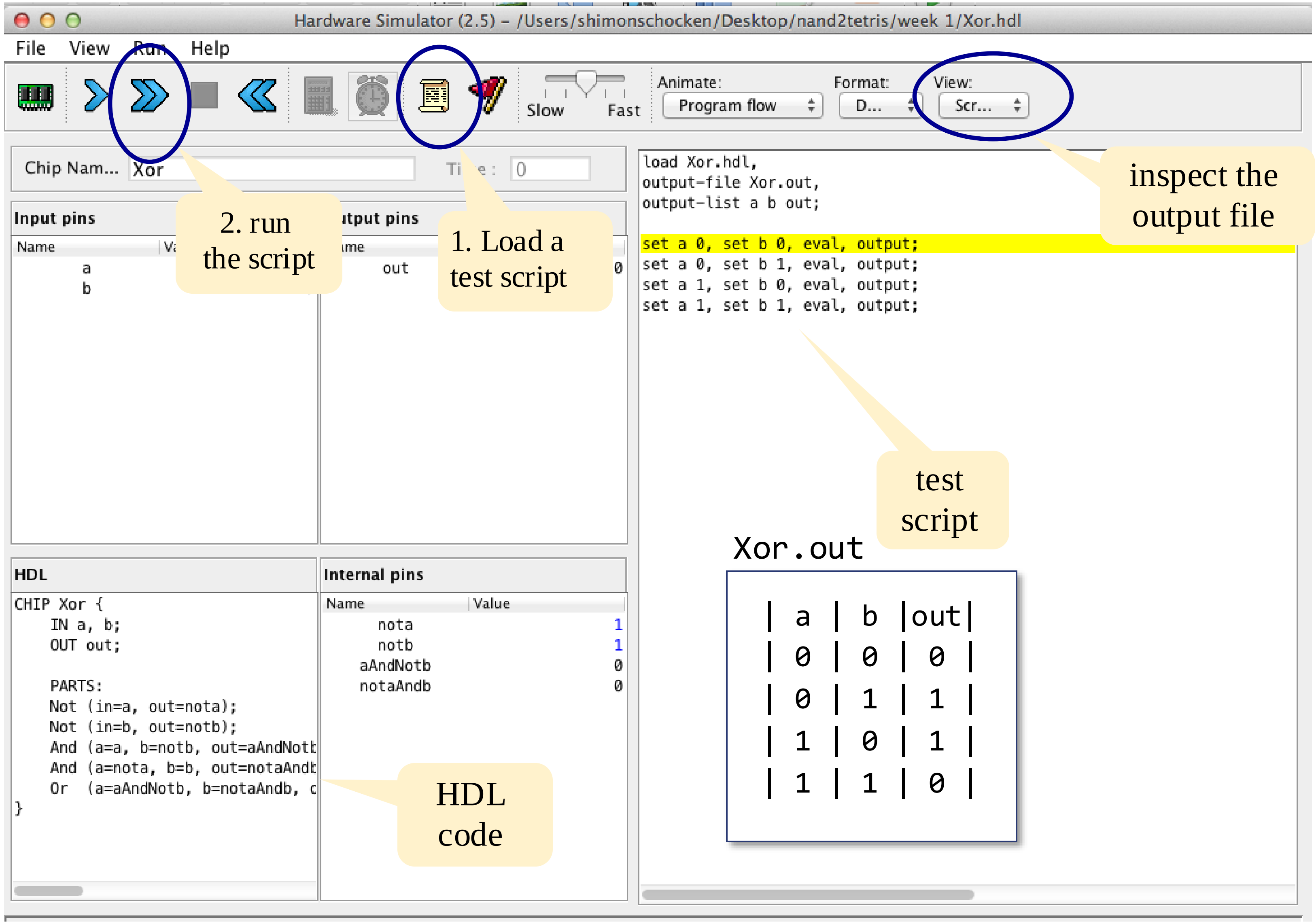
```
| a | b |out|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |
```

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` (inputs) , `eval` (chip logic) , `output` (print)

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
- 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

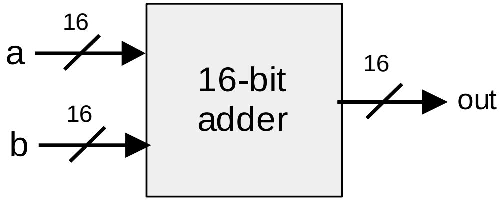
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	0	0	1	0	0	0	1	1	0	1	1	1	0	1

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:
   ...
}
```

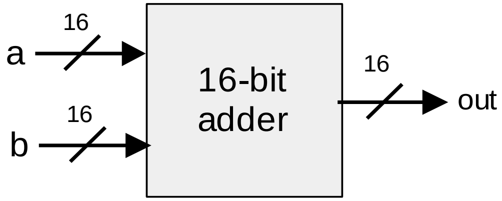


	15	…	1	0
a:	1	...	1	1
b:	0	...	1	0
c:	0	...	0	1
out:	1	...	1	0

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

   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;
   ...
}
```

	3	2	1	0
a:	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;
   ...
}
```

	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=      , 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;
   ...
}
```

```
/* 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	1	1

out: 0

```
/* 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

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

More on busses:

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

<u>Behavioral simulation</u>

- 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:

- 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`

stub file

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

    PARTS:
    // Implementation missing
}
```

`Xor.tst`

test script

```
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;
```

compare file

`Xor.cmp`

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

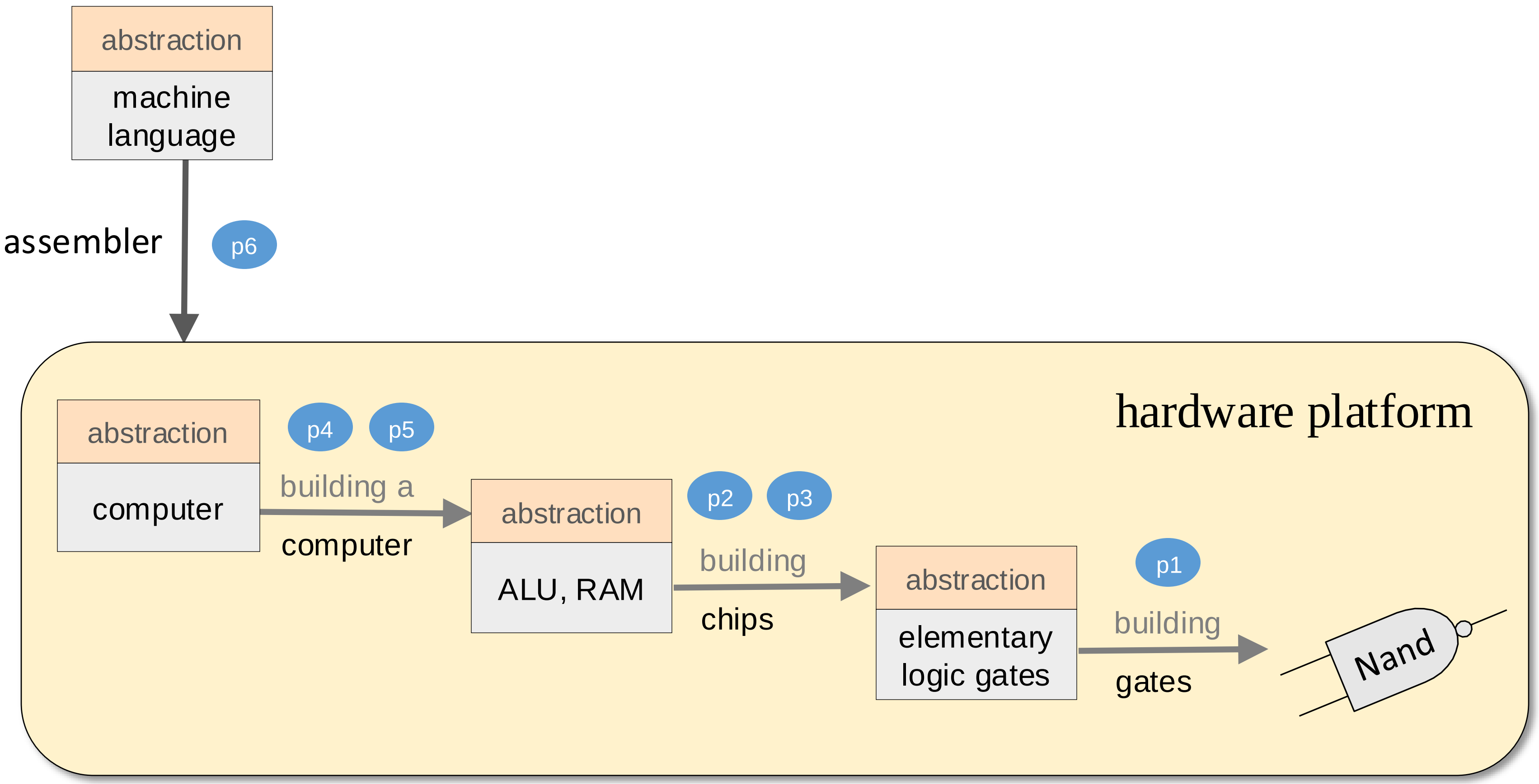
These files specify:

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

The developer's task:

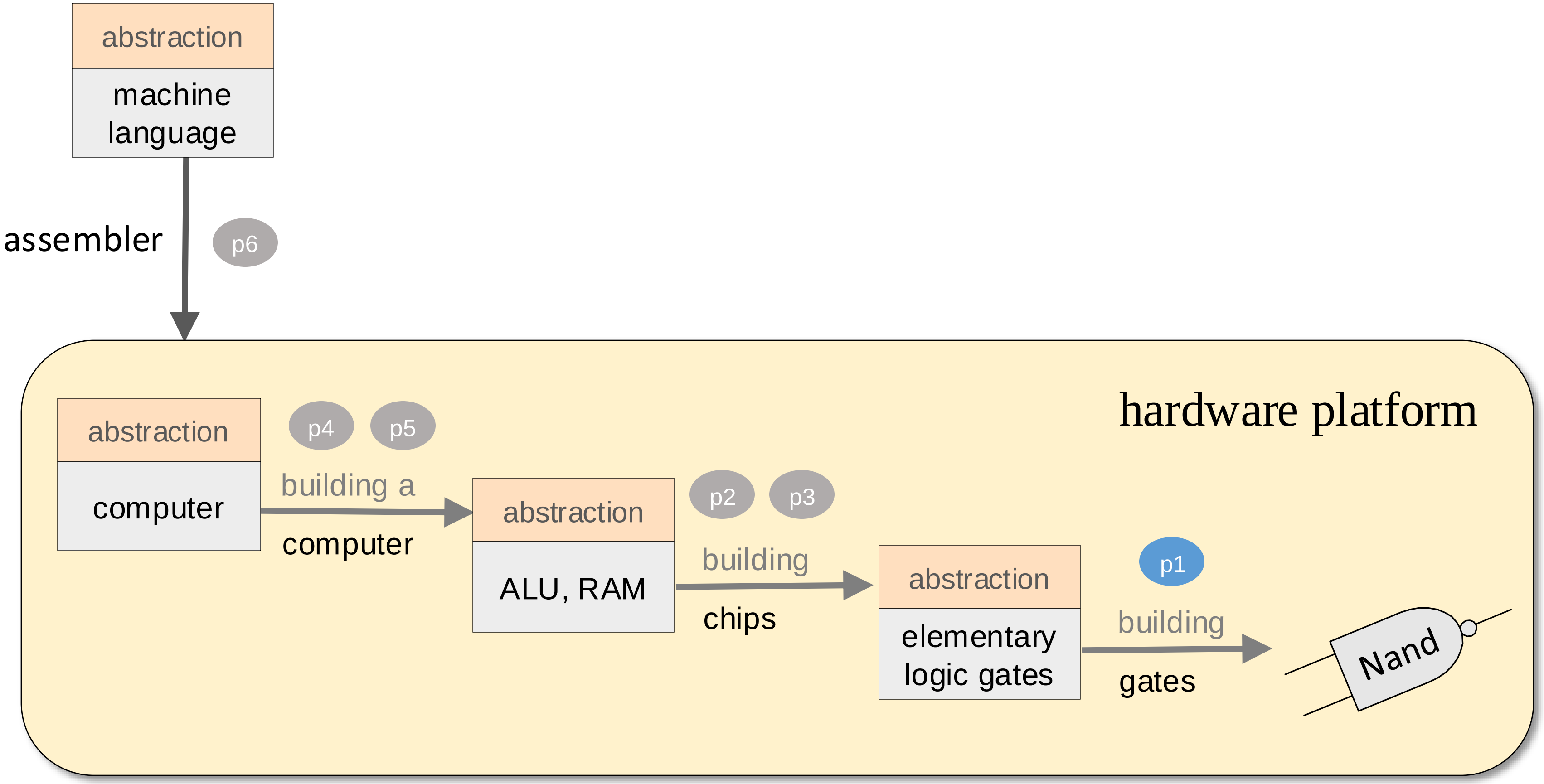
Implement the chip (complete the supplied `.hdl` file), using these resources.

Project 1



abstraction
machine
language
assembler
p6
hardware platform
abstraction
computer
p4
p5
building a
computer
abstraction
ALU, RAM
p2
p3
building
chips
abstraction
elementary
logic gates
p1
building
gates
Nand

Project 1



Project 1

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

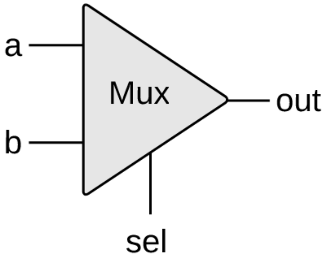
16-bit variants

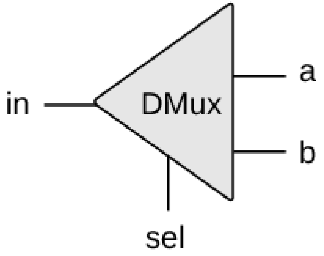
- Not16
- And16
- Or16
- Mux16

Multi-way variants

- Or8Way
- Mux4Way16
- Mux8Way16
- DMux4Way
- DMux8Way

Multiplexor / Demultiplexor





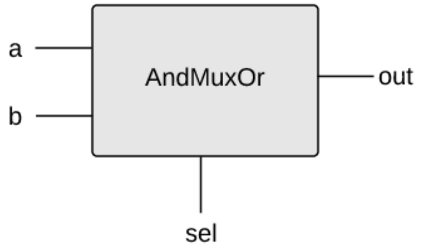

```
if (sel == 0)
   out = a
else
   out = b
```

```
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

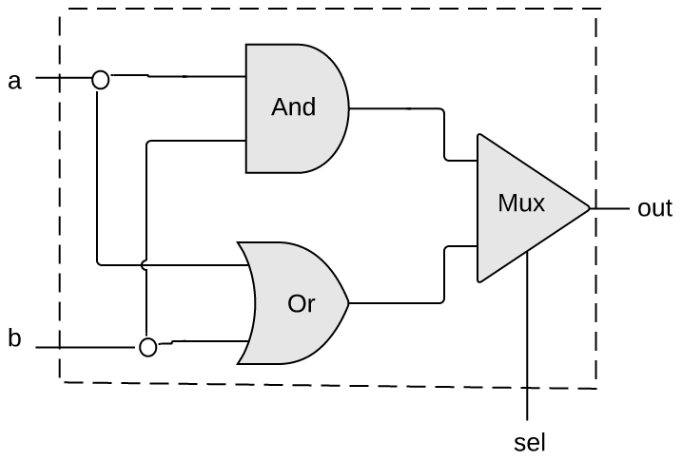


if (`sel == 0`)
    `out = a And b`
else
    `out = a Or b`

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

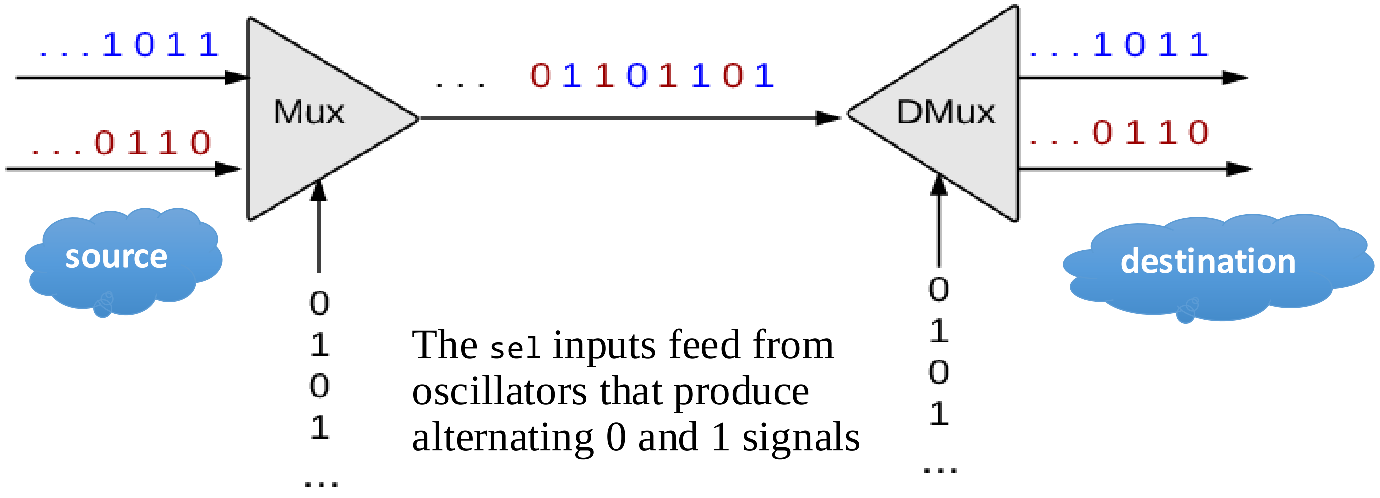
When `sel == 1` the gate acts like an Or gate



Mux.hdl

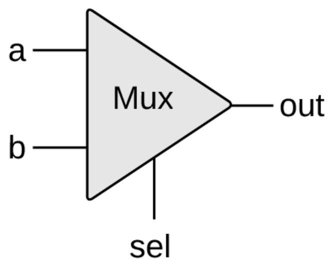
```
CHIP AndMuxOr {
    IN a, b, sel;
    OUT out;

    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

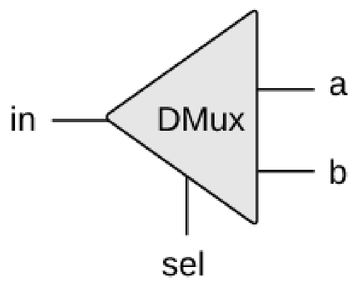
`Mux.hdl`

```
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:
}
```

<u>Implementation tip</u>

Simple truth table, simple implementation.

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

Project 1

<u>Elementary logic gates</u>

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

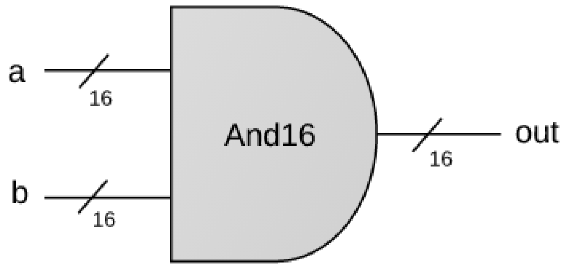
<u>16-bit variants</u>

- Not16
- And16
- Or16
- Mux16

<u>Multi-way variants</u>

- Or8Way
- Mux4Way16
- Mux8Way16
- DMux4Way
- DMux8Way

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:
}
```

Implementation tip

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`

Project 1

Elementary logic gates

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

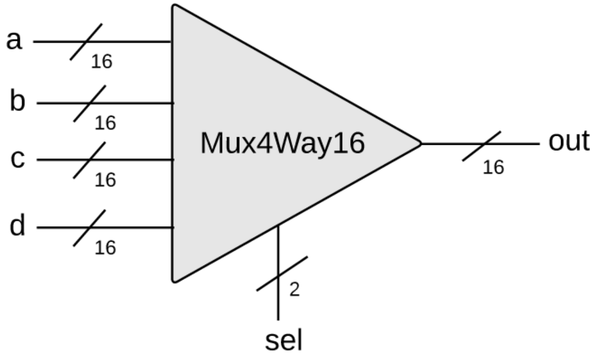
16-bit variants

- Not16
- And16
- Or16
- Mux16

Multi-way variants

- Or8Way
- Mux4Way16
- Mux8Way16
- DMux4Way
- DMux8Way

16-bit, 4-way multiplexor



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

Mux4Way16.hdl

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

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

<u>Implementation tip:</u>

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

<u>Elementary logic gates</u>

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

<u>16-bit variants</u>

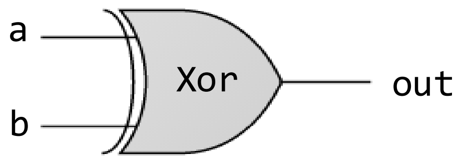
- Not16
- And16
- Or16
- Mux16

<u>Multi-way variants</u>

- Or8Way
- Mux4Way16
- Mux8Way16
- DMux4Way
- DMux8Way

How to actually <u>build</u> 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

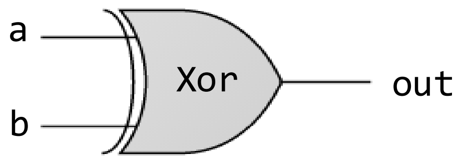
Xor.hdl (stub file)

```
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
```

<u>The contract:</u>

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;
```

<u>Project 1 folder</u>

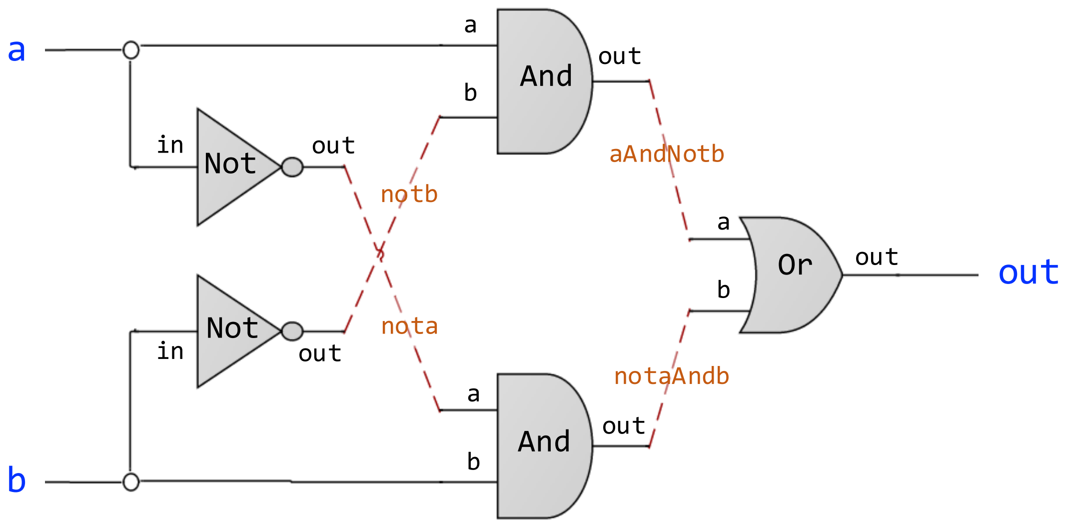
(`.hdl`, `.tst`, `.cmp` files):

`nand2tetris/projects/1`

<u>Tools:</u>

- 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-part, say `Bar`:

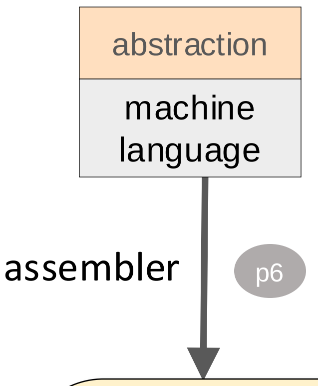
If using the web-based IDE:

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

If using the desktop hardware simulator:

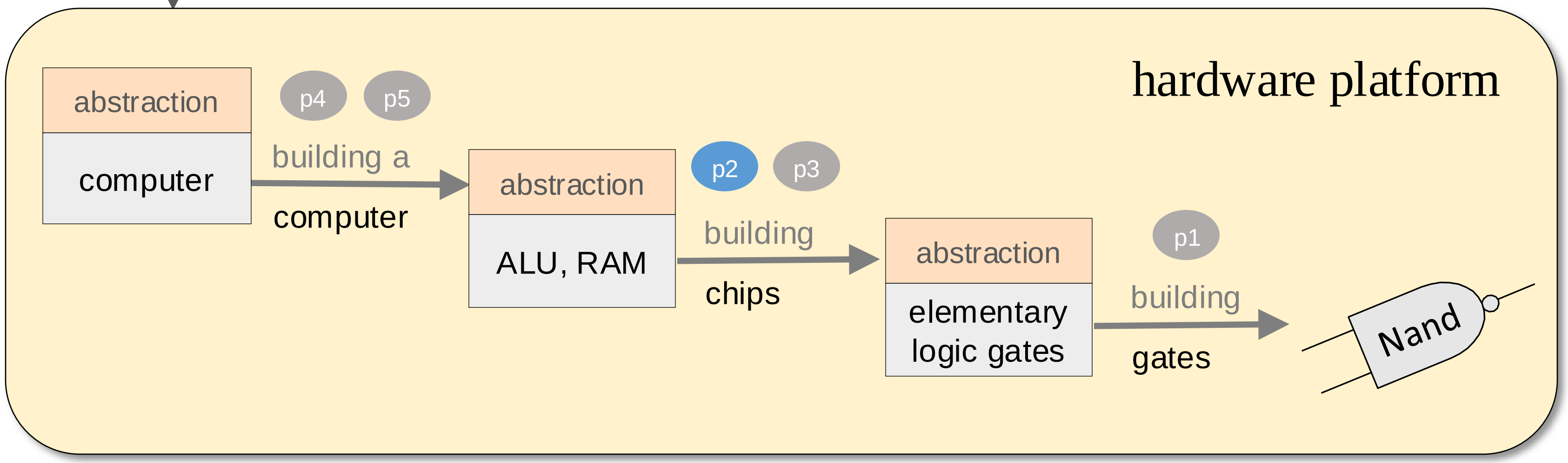
- Typically, `Bar.hdl` is a stub-file, or a file that has an incomplete implementation
- If you want to use `Bar` as a built-in chip:
  Remove the file `Bar.hdl` from the project folder (or rename it, say, `Bar1.hdl`)
- The result: Whenever `Bar` will be mentioned as a chip-part in some chip definition,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



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