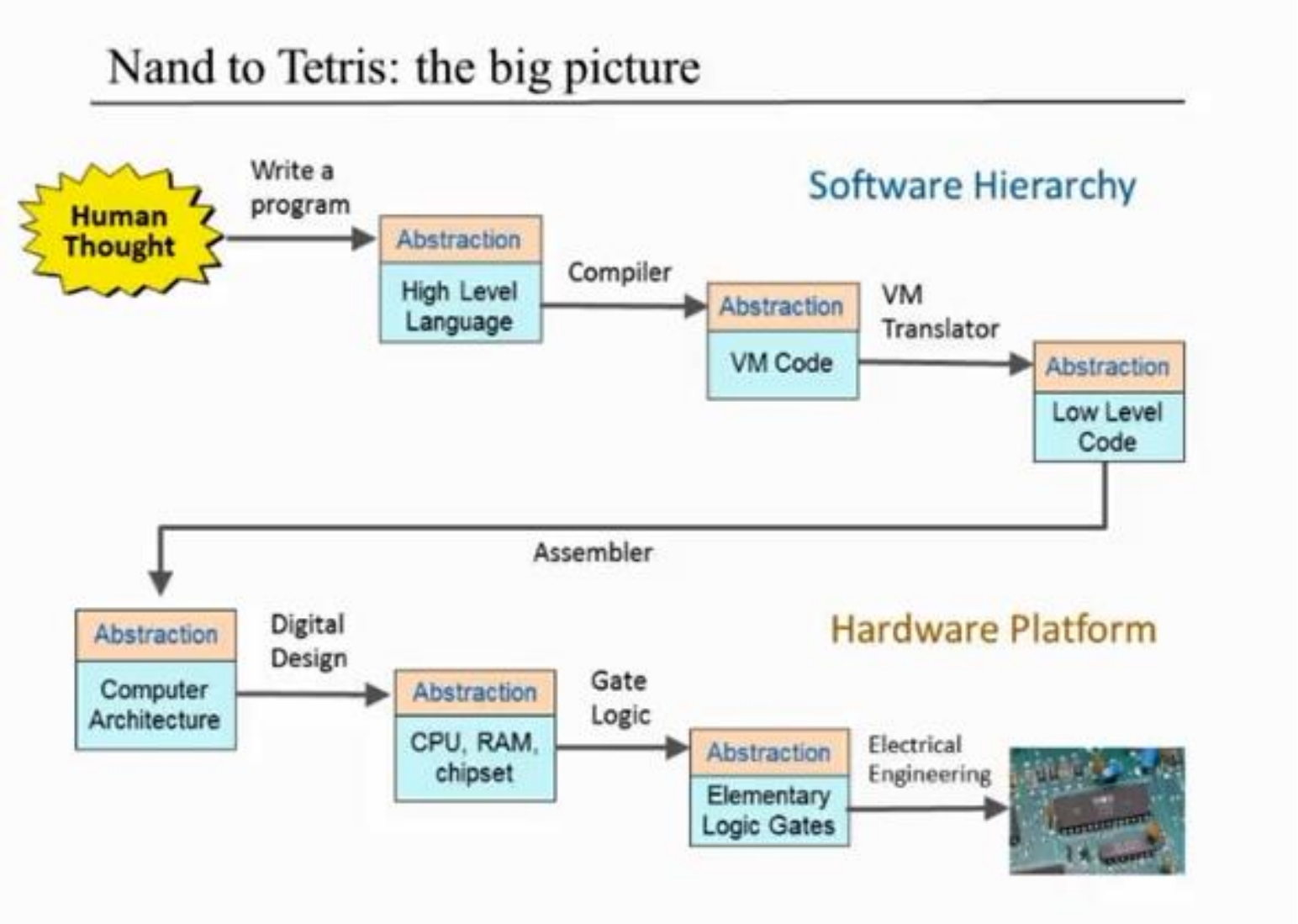
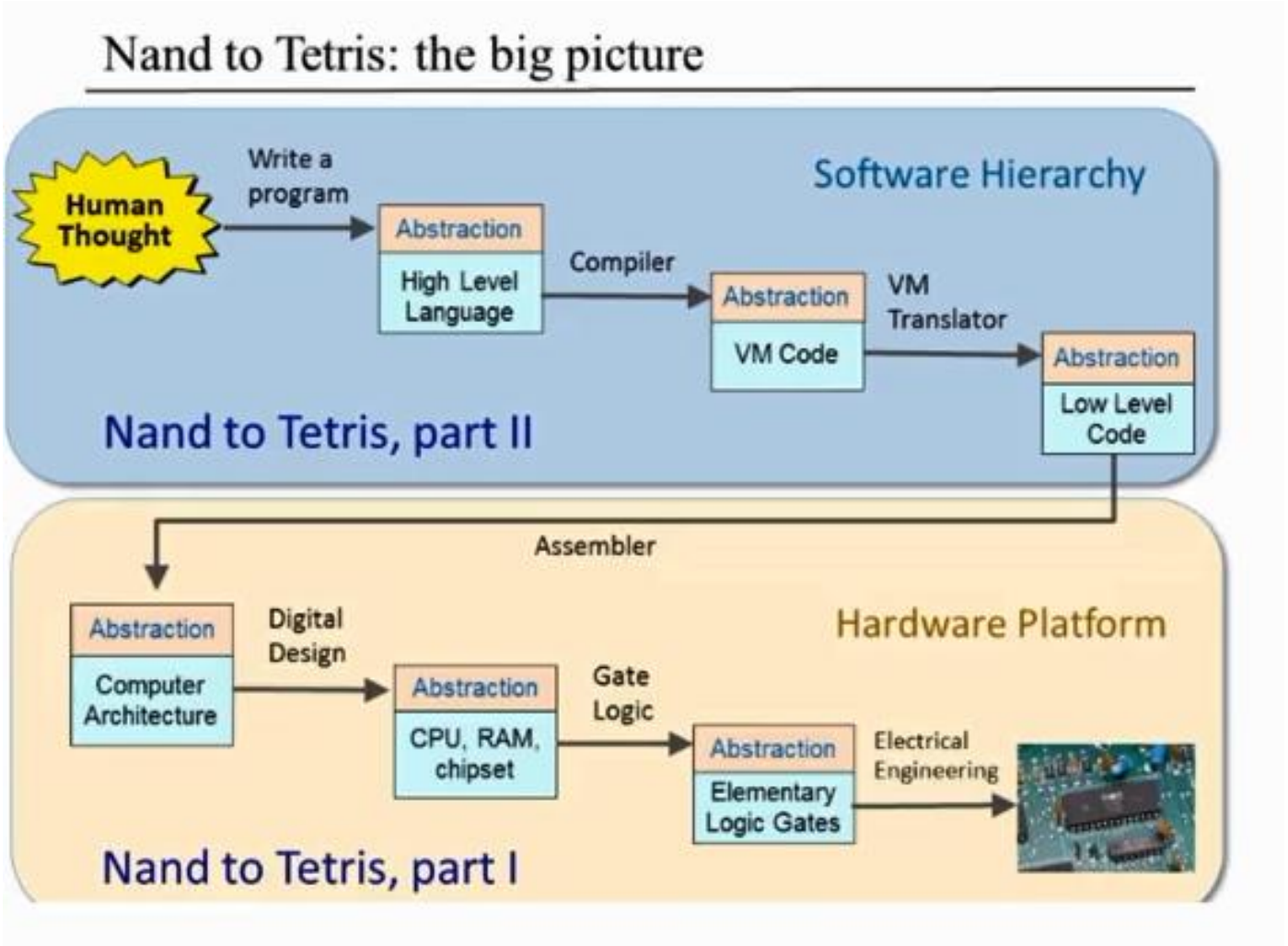
Computer Architecture

Big Picture

NAND TO APPS


Nand to Tetris: the big picture
Human Thought
Write a program
Abstraction
High Level Language
Compiler
Abstraction
VM Code
VM Translator
Abstraction
Low Level Code
Software Hierarchy
Assembler
Abstraction
Computer Architecture
Digital Design
Abstraction
CPU, RAM, chipset
Gate Logic
Abstraction
Elementary Logic Gates
Electrical Engineering
Hardware Platform

NAND TO APPS

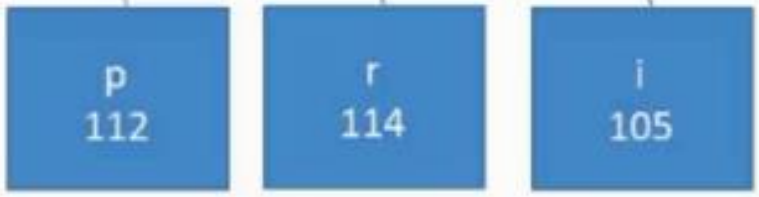

Nand to Tetris: the big picture
Human Thought
Write a program
Abstraction
High Level Language
Compiler
Abstraction
VM Code
VM Translator
Abstraction
Low Level Code
Software Hierarchy
Nand to Tetris, part II
Assembler
Abstraction
Computer Architecture
Digital Design
Abstraction
CPU, RAM, chipset
Gate Logic
Abstraction
Elementary Logic Gates
Electrical Engineering
Hardware Platform
Nand to Tetris, part I

Hello World

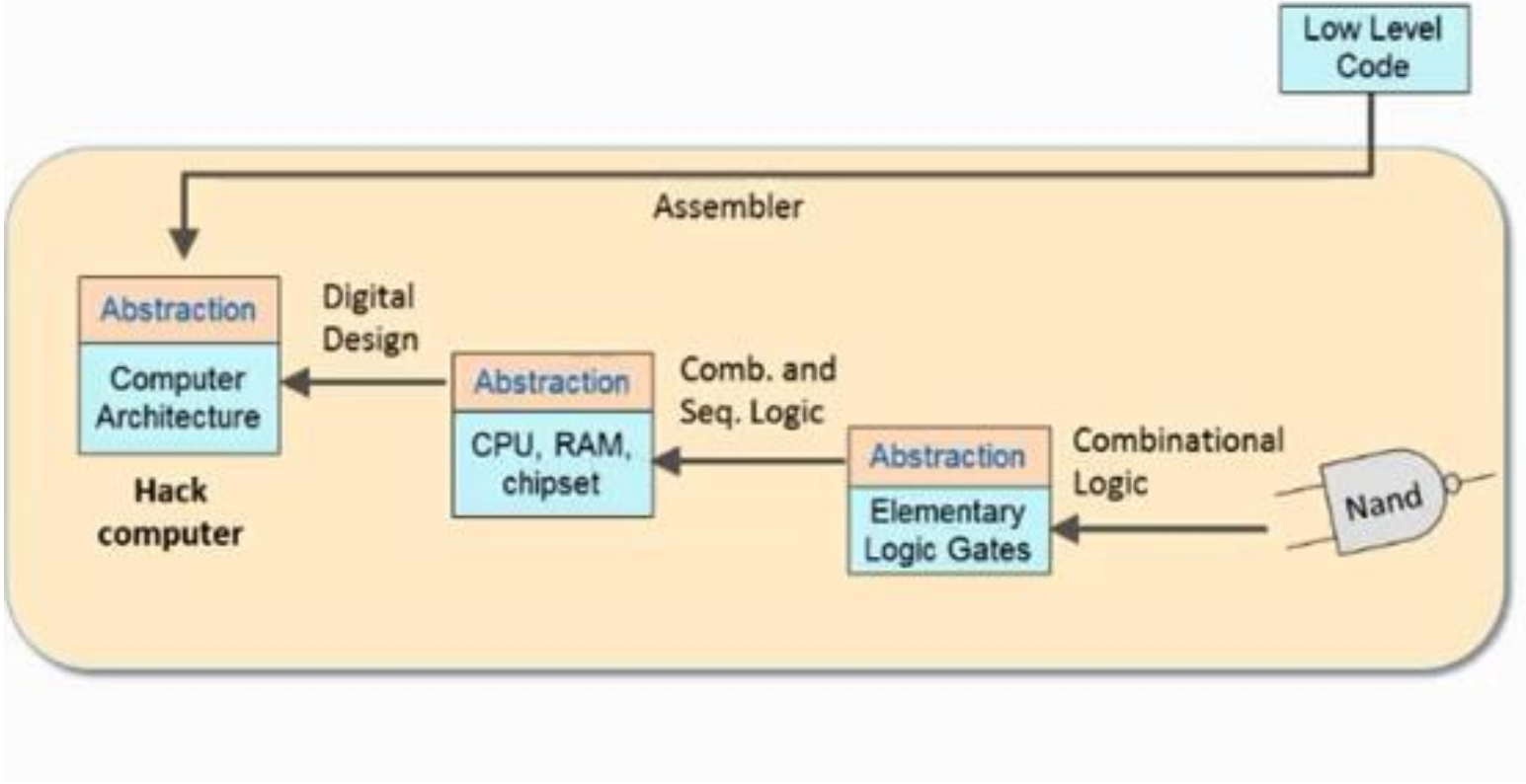
```
// First Example in Programming 101
Class Main {
    function void main() {
        do Output.printString("Hello World!");
        do Output.println(); // New line
        return;
    }
}
```

How do these letters do anything?

```
// First Example in Programming 101
Class Main {
    function void main() {
        do Output.printString("Hello World!");
        do Output.println(); // New line
        return;
     }
}
```



Nand to Tetris, Part I



Hardware Part

Building a chip: design

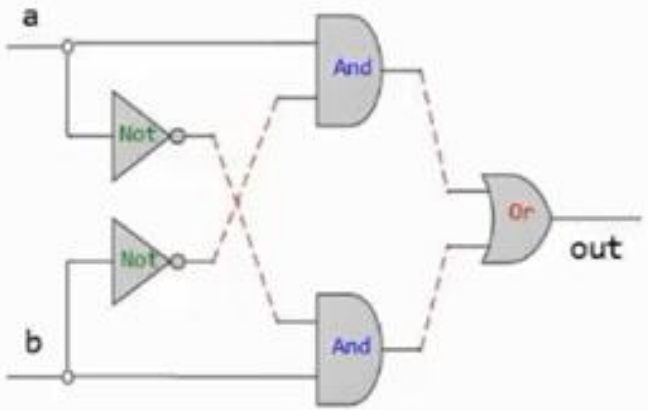
chip abstraction

Xor: outputs 1 if exactly one of its inputs is 1, otherwise outputs 0.

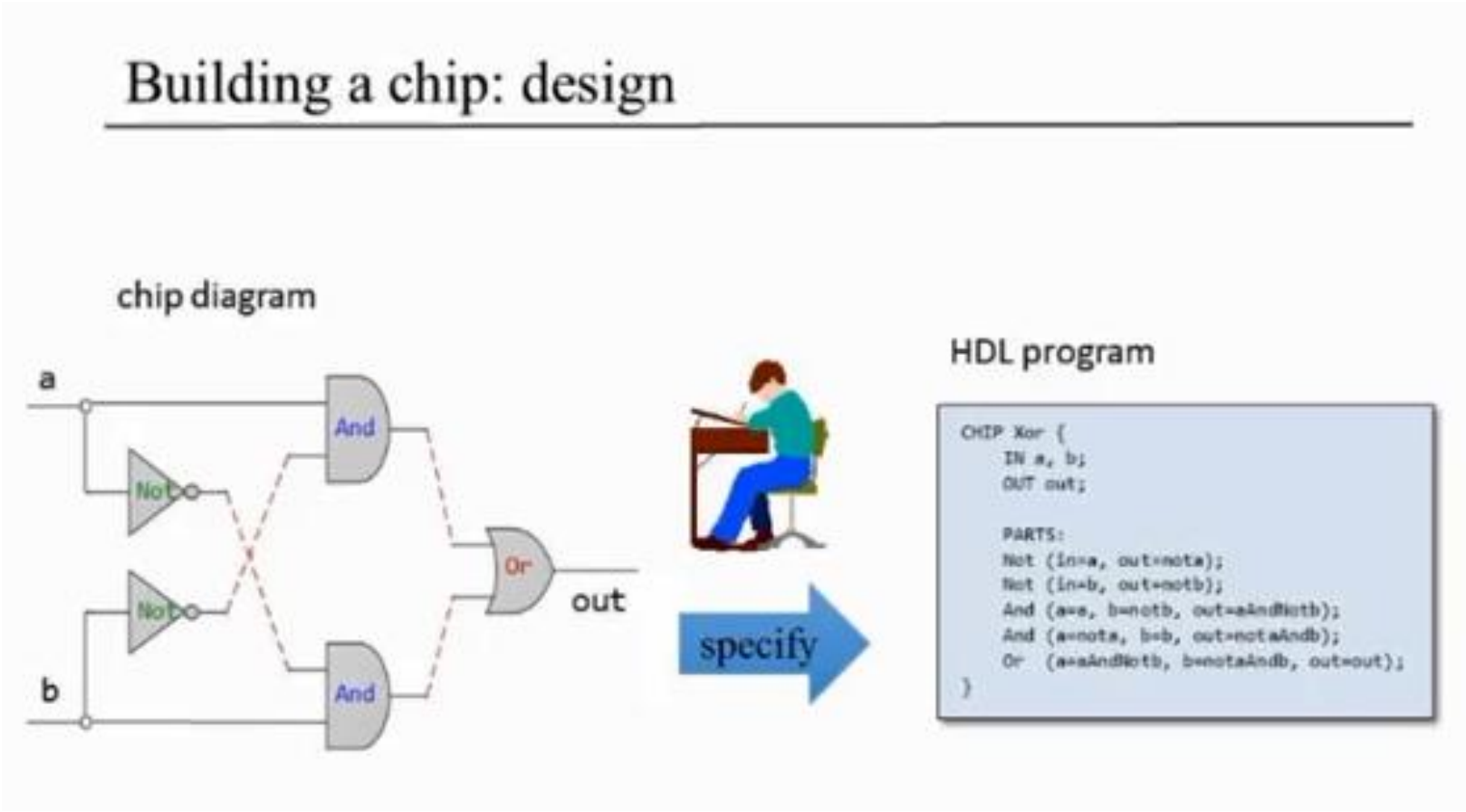
a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

design

chip diagram



Hardware Part

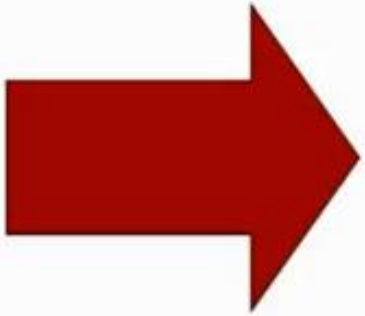


Nand to Tetris, Part I

You have built a complete functioning computer

- That can run anything including games like Tetris
- Using only modest Nand gates

a	b	Nand
0	0	1
0	1	1
1	0	1
1	1	0



Low level languages

Programming at the end of Part I: Hack

Hack assembly code
(source language)

```
// Computes RAM[1]=1+...+RAM[0]
    @i
    M=1   // i = 1
    @sum
    M=0   // sum = 0

(LOOP)
    @i    // if i>RAM[0] goto STOP
    D=M
    @R0
    D=D-M
    @STOP
    D;JGT
    @i    // sum += i
    D=M
    @sum
    M=D+M
    @i    // i++
    M=M+1
    @LOOP // goto LOOP
    0;JMP
    ...
```

Hack binary code

```
0000000000010000
1110111111001000
0000000000010001
1110101010001000
0000000000010000
1111110000010000
0000000000000000
1111010011010000
0000000000010010
1110001100000001
0000000000010000
1111110000010000
0000000000010001
1111000010001000
0000000000010000
1111110111001000
0000000000000100
1110101010000111
...
```

- How can we construct hardware units for basic operations?

Boolean Logic

Boolean Values

F	T
N	Y
0	1

Boolean Operation

(x AND y)

$x \wedge y$

x	y	AND
0	0	0
0	1	0
1	0	0
1	1	1

(x OR y)

$x \vee y$

x	y	OR
0	0	0
0	1	1
1	0	1
1	1	1

NOT(x)

$\neg x$

x	NOT
0	1
1	0

NOT(0 OR (1 AND 1)) = NOT(0 OR 1) = NOT(1) = 0

Boolean Identities

Boolean Identities

- (x AND y) = (y AND x)
- (x OR y) = (y OR x)

Commutative Laws

- (x AND (y AND z)) = ((x AND y) AND z)
- (x OR (y OR z)) = ((x OR y) OR z)

Associative Laws

- (x AND (y OR z)) = (x AND y) OR (x AND z)
- (x OR (y AND z)) = (x OR y) AND (x OR z)

Distributive Laws

- NOT(x AND y) = NOT(x) OR NOT(y)
- NOT(x OR y) = NOT(x) AND NOT(y)

De Morgan Laws

De Morgans Teorem

$(X + Y)' = X' \cdot Y'$

NOR is equivalent to AND with inputs complemented

X	Y	X'	Y'	(X + Y)'	X' • Y'
0	0	1	1	1	1
0	1	1	0	0	0
1	0	0	1	0	0
1	1	0	0	0	0

$(X \cdot Y)' = X' + Y'$

NAND is equivalent to OR with inputs complemented

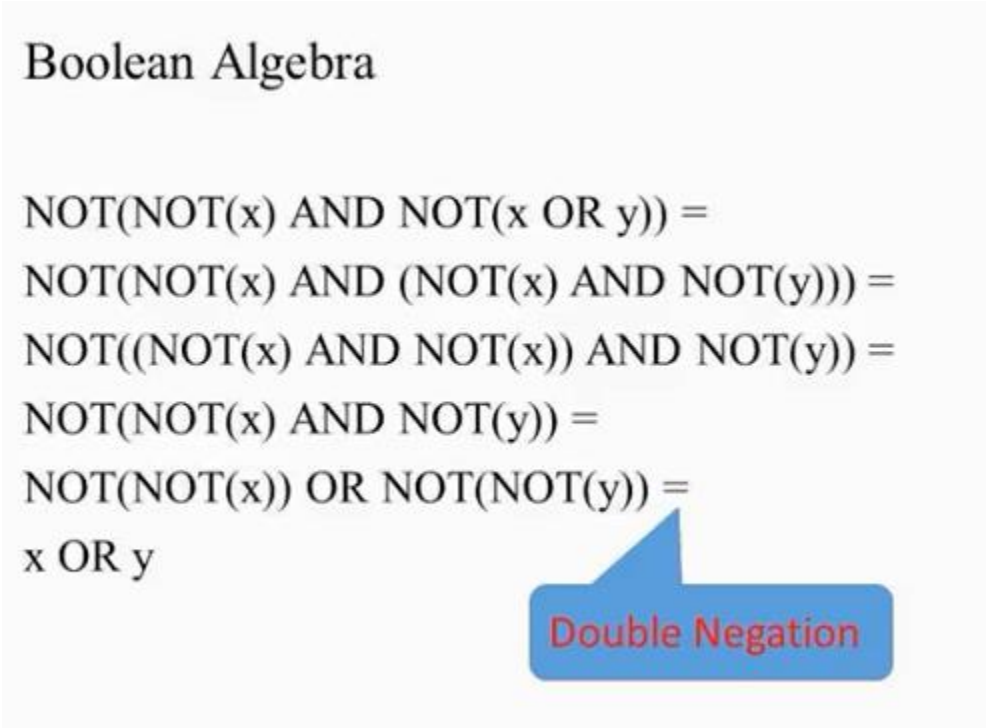
X	Y	X'	Y'	(X • Y)'	X' + Y'
0	0	1	1	1	1
0	1	1	0	1	1
1	0	0	1	1	1
1	1	0	0	0	0

- Example: Find the complement of $Z = A'B'C + A'BC + AB'C + ABC'$

$$Z' = (A'B'C + A'BC + AB'C + ABC')'$$
$$= (A'B'C)' \cdot (A'BC)' \cdot (AB'C)' \cdot (ABC')'$$
$$= (A+B+C') \cdot (A+B'+C') \cdot (A'+B+C') \cdot (A'+B'+C)$$

Boolean Algebra

- Simplifing boolean expression via boolean algebra



- Same conclusion with truth table..

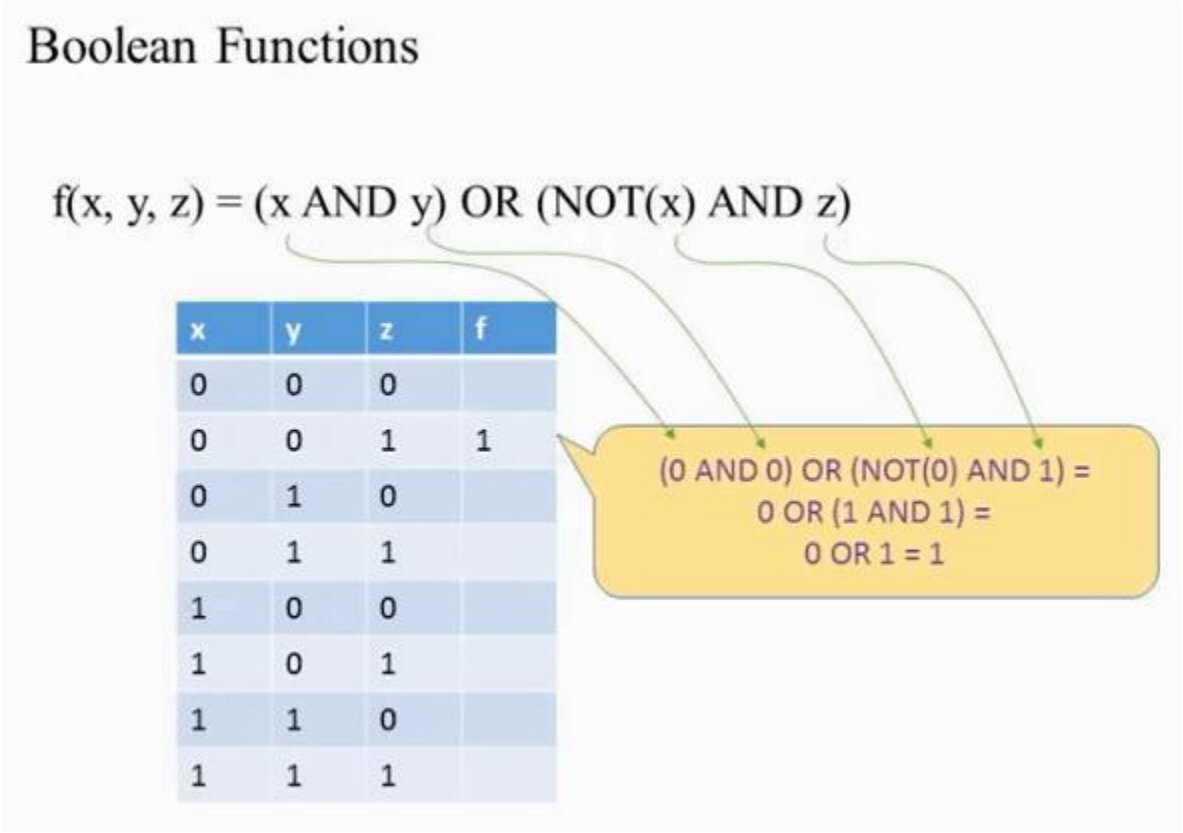
Boolean Algebra

NOT(NOT(x) AND NOT(x OR y)) =

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

= x OR y

Boolean Function



Boolean Function

Boolean Functions

$f(x, y, z) = (x \text{ AND } y) \text{ OR } (\text{NOT}(x) \text{ AND } z)$ — Formula

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

Truth table

Truth Table to Boolean Expression

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

(NOT(x) AND NOT(y) AND NOT(z))

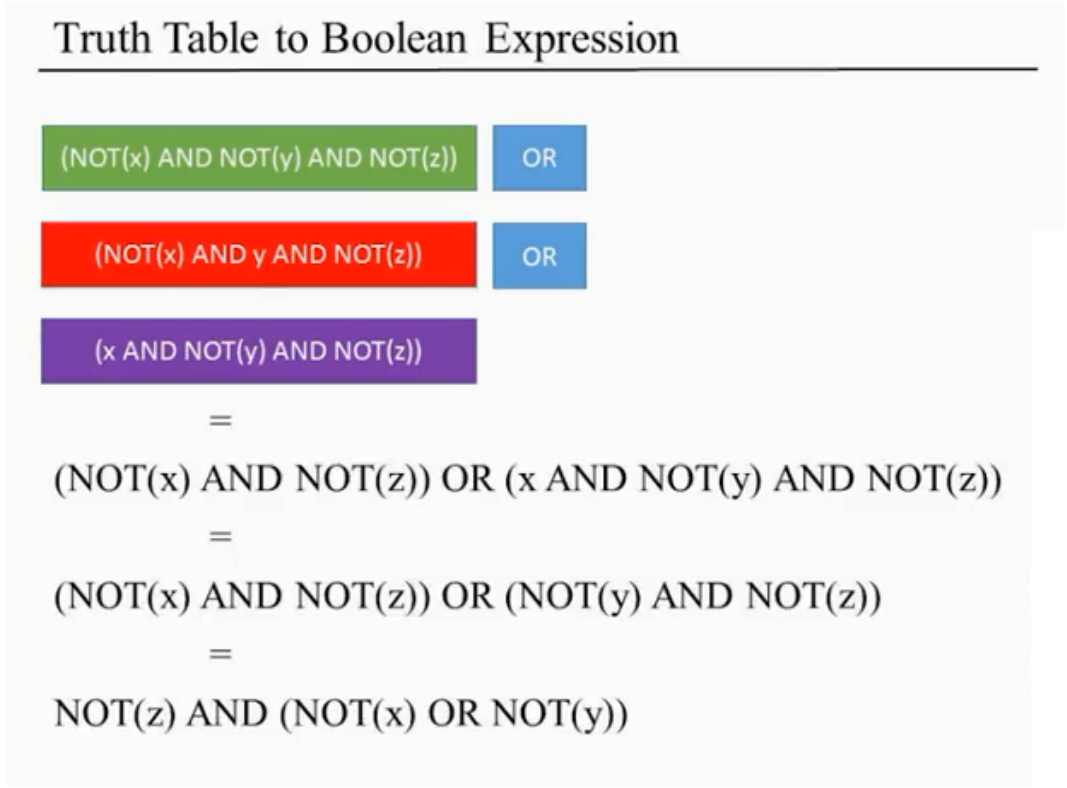
OR

(NOT(x) AND y AND NOT(z))

OR

(x AND NOT(y) AND NOT(z))

- Finding the shortest expression NP hard problem



Theorem

Any Boolean function can be represented using an expression containing AND, OR and NOT operations.

Theorem

Any Boolean function can be represented using an expression containing AND and NOT operations.

Proof:

$$(x \text{ OR } y) = \text{NOT}(\text{NOT}(x) \text{ AND } \text{NOT}(y))$$

Question

What would be the logical equivalent of NAND(x,x)?

Remember NAND(x,x) is defined to be NOT(x AND x). Here is the truth table.

x	y	NAND
0	0	1
0	1	1
1	0	1
1	1	0

- x
- OR(x,x)
- AND(x,x)
- NOT(x)

Theorem

Any Boolean function can be represented using an expression containing only NAND operations.

Proof:

1) NOT(x) = (x NAND x)

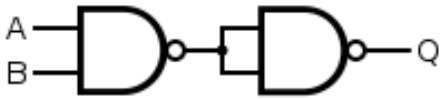
2) (x AND y) = NOT(x NAND y)

3) (x OR y)= NOT(x) NAND NOT(y)

Desired AND Gate

Q = **A** AND **B**

NAND Construction



= (**A** NAND **B**) NAND (**A** NAND **B**)

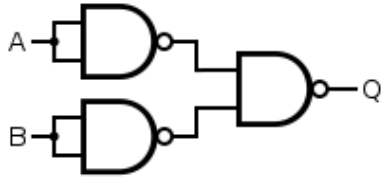
Truth Table

Input A	Input B		Output Q
0	0		0
0	1		0
1	0		0
1	1		1

Desired OR Gate

Q = **A** OR **B**

NAND Construction



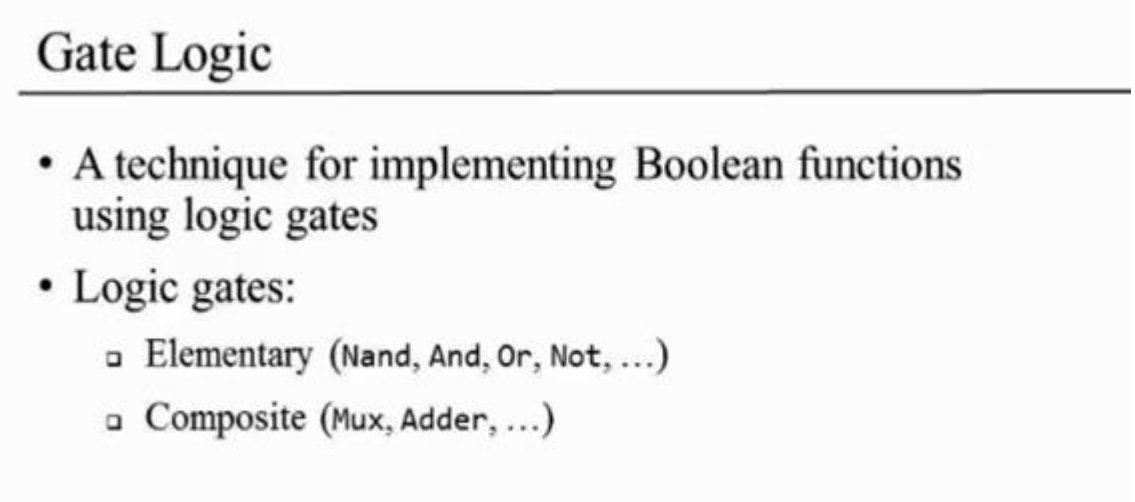
= (**A** NAND **A**) NAND (**B** NAND **B**)

Truth Table

Input A	Input B		Output Q
0	0		0
0	1		1
1	0		1
1	1		1

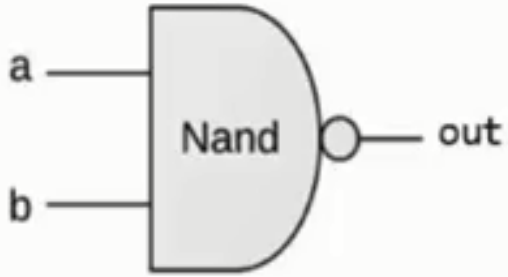
Logic Gates

- Now we are going from abstract boolean logic to computer parts



Elementary logic gates: Nand

gate diagram:



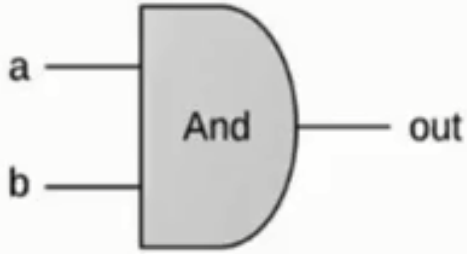
functional specification:

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

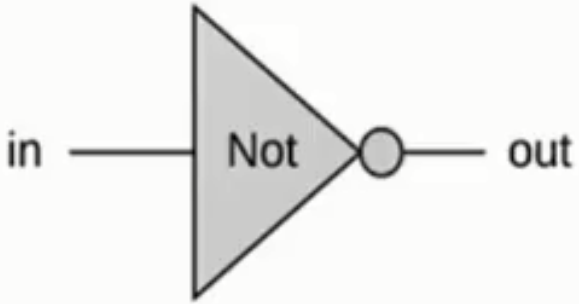
truth table:

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

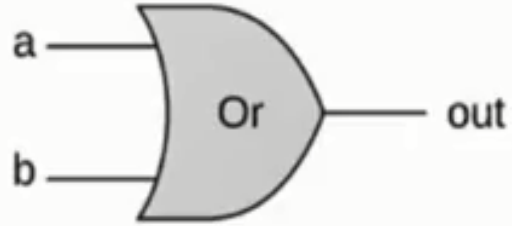
Elementary logic gates: And, Or, Not




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

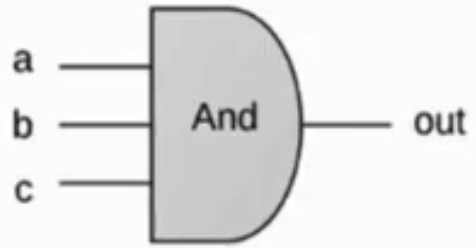



```
if (in==0)
then out=1 else out=0
```

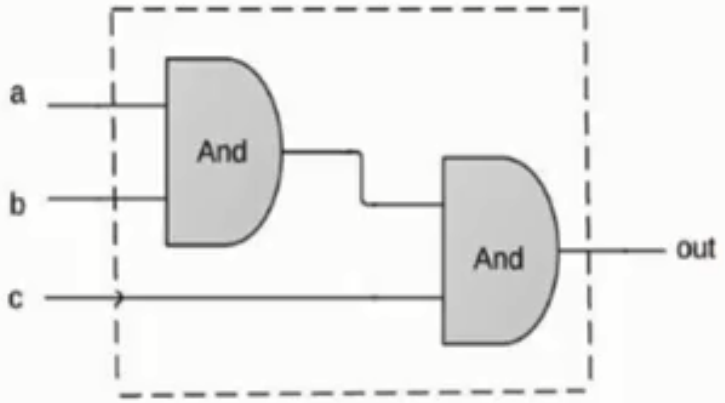



```
if (a==1 or b==1)
then out=1 else out=0
```

Composite gates

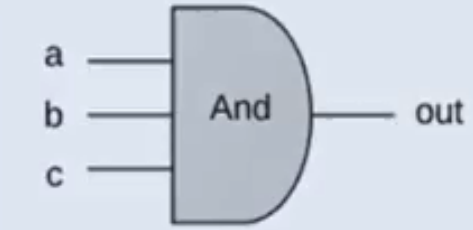



```
if (a==1 and b==1 and c==1)
   then out=1 else out=0
```

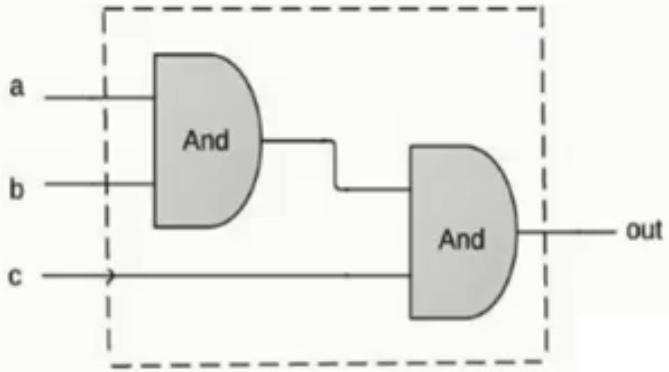


Gate Interface / Gate Implementation

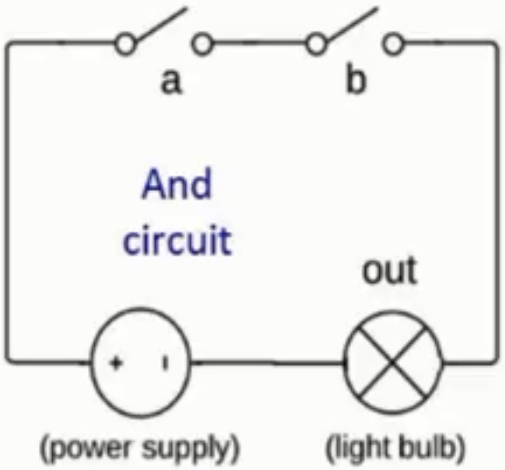
gate interface



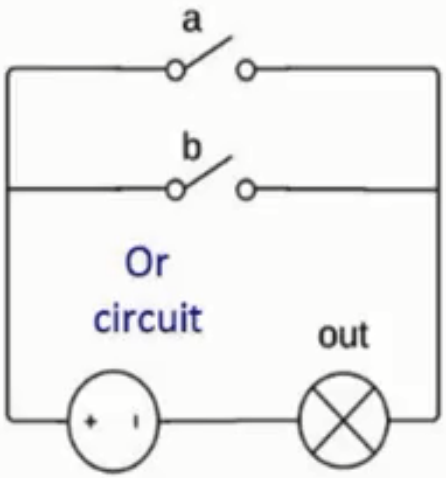

```
if (a==1 and b==1 and c==1)
   then out=1 else out=0
```



Circuit implementations



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



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

- HARDWARE DESCRIPTION LANGUAGES

Hardware Part

Building a chip: design

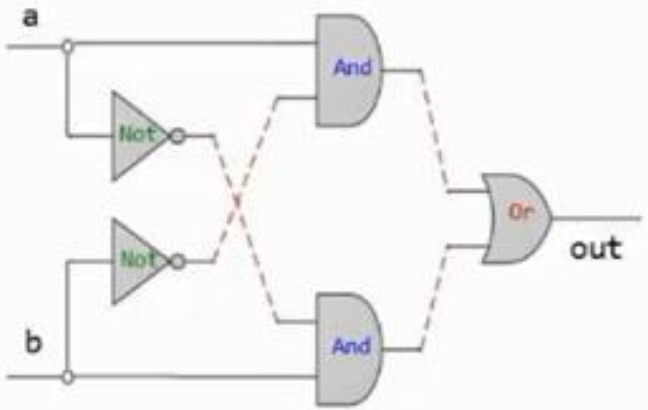
chip abstraction

Xor: outputs 1 if exactly one of its inputs is 1, otherwise outputs 0.

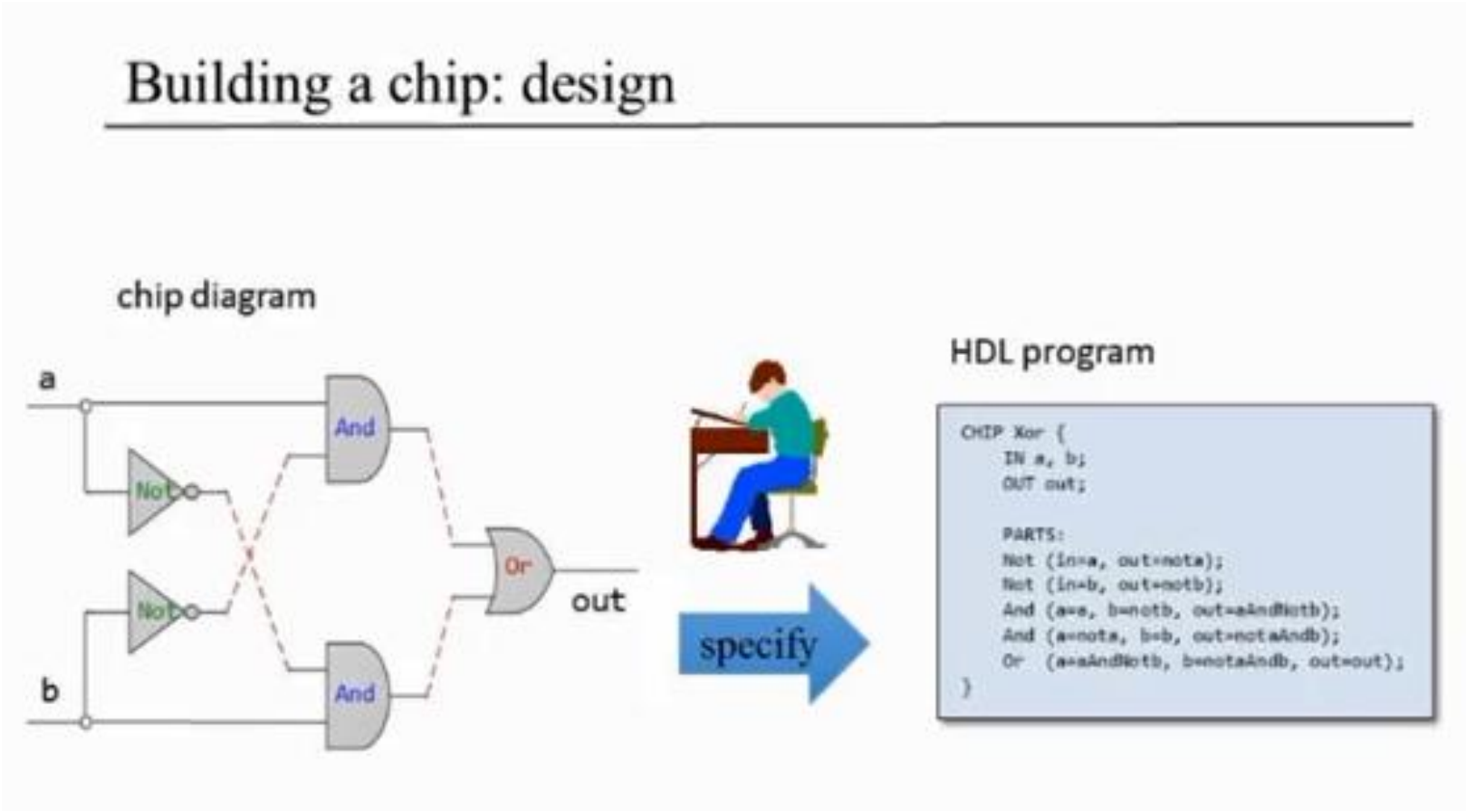
a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

design

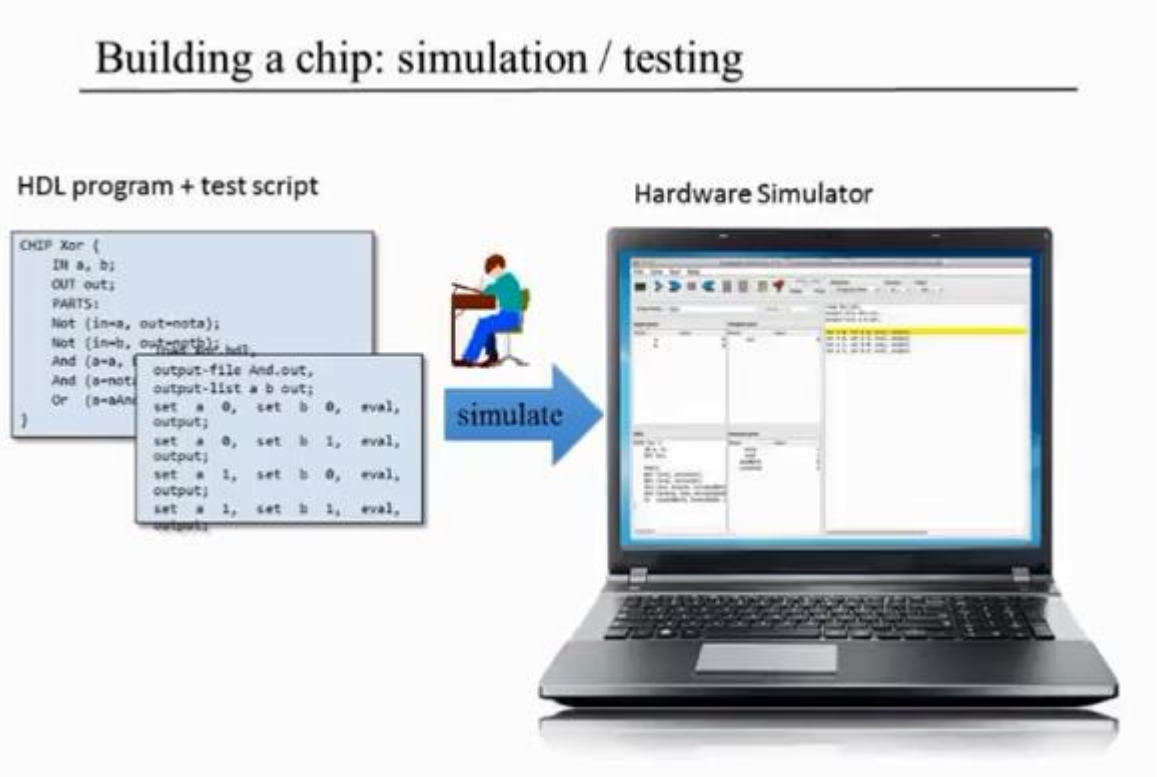
chip diagram



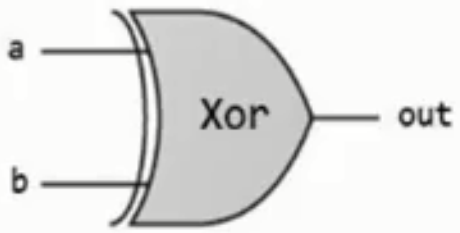
Hardware Part



- We can build the chip via using computers



Design: from requirements to interface



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

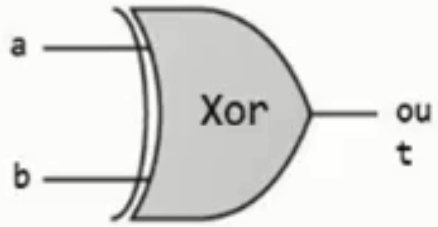
a	b	out
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

Requirement:

Build a gate that delivers this functionality

```
/** Xor gate: out = (a And Not(b)) Or (Not(a) And b)) */

CHIP Xor {
    IN a, b;
    OUT out;

    PARTS:
    // Implementation missing
}
```

Design: from requirements to gate diagram



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

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

Requirement:

Build a gate that delivers this functionality

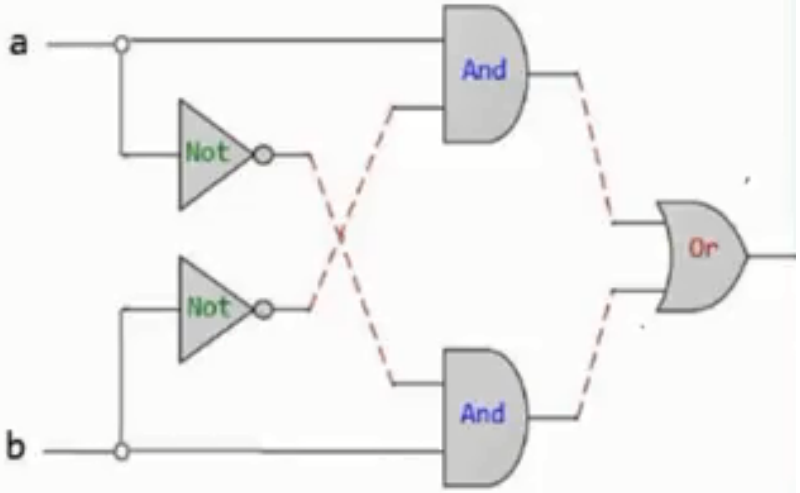
General idea:

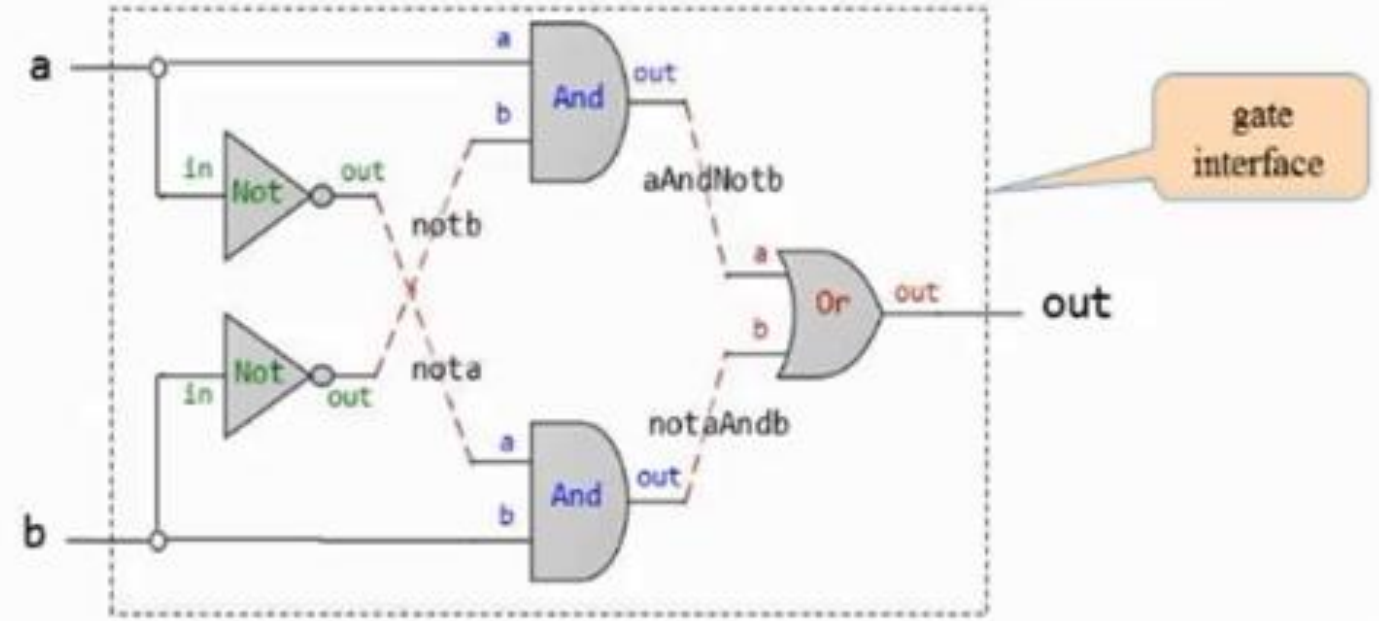
`out=1` when:

`a` And Not(`b`)

Or

`b` And Not(`a`)





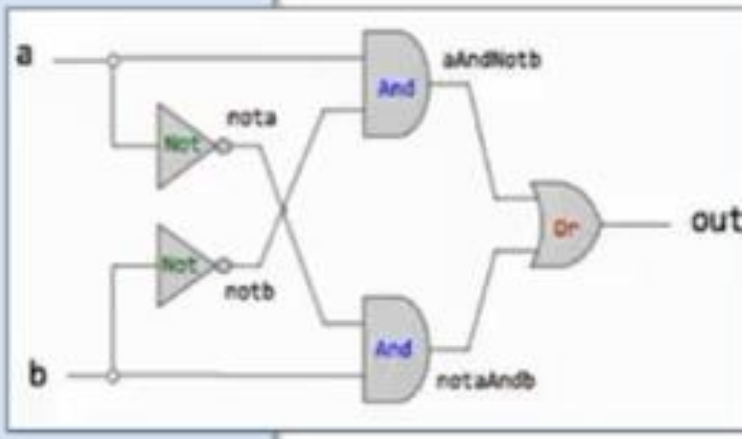
interface

```
/** Xor gate: out = (a And Not(b)) Or (Not(a) And b))   */

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

implementation

```
    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);
}
```

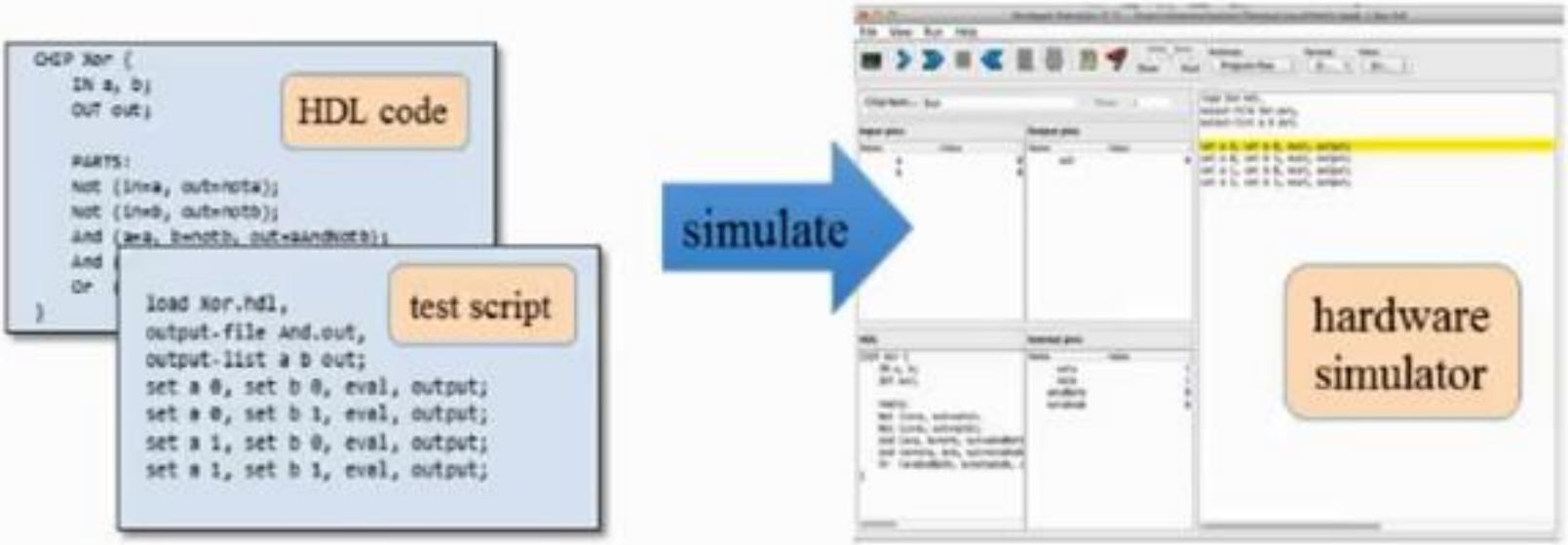
```
/** Xor gate: out = (a And Not(b)) Or (Not(a) And b))   */

CHIP Xor {
    IN a, b;
    OUT out;

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



- HDL is a functional / declarative language
- The order of HDL statements is insignificant
- Before using a chip part, you must know its interface. For example:

  `Not(in= ,out=), And(a= ,b= ,out= ), Or(a= ,b= ,out= )`

Hardware simulation in a nutshell



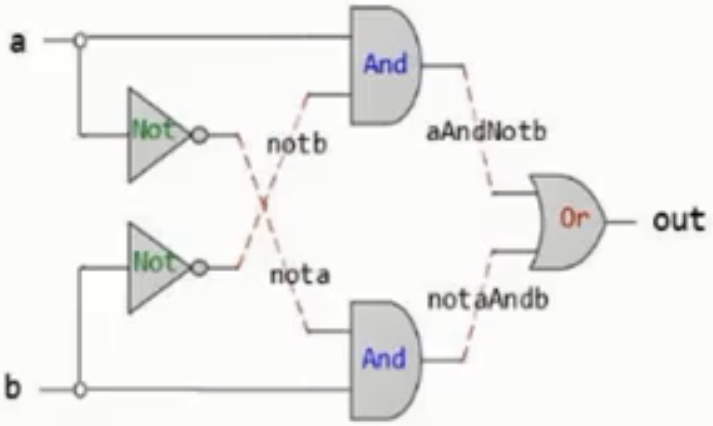
Simulation options:

- Interactive

Interactive simulation (using Xor as an example)

Xor.hdl

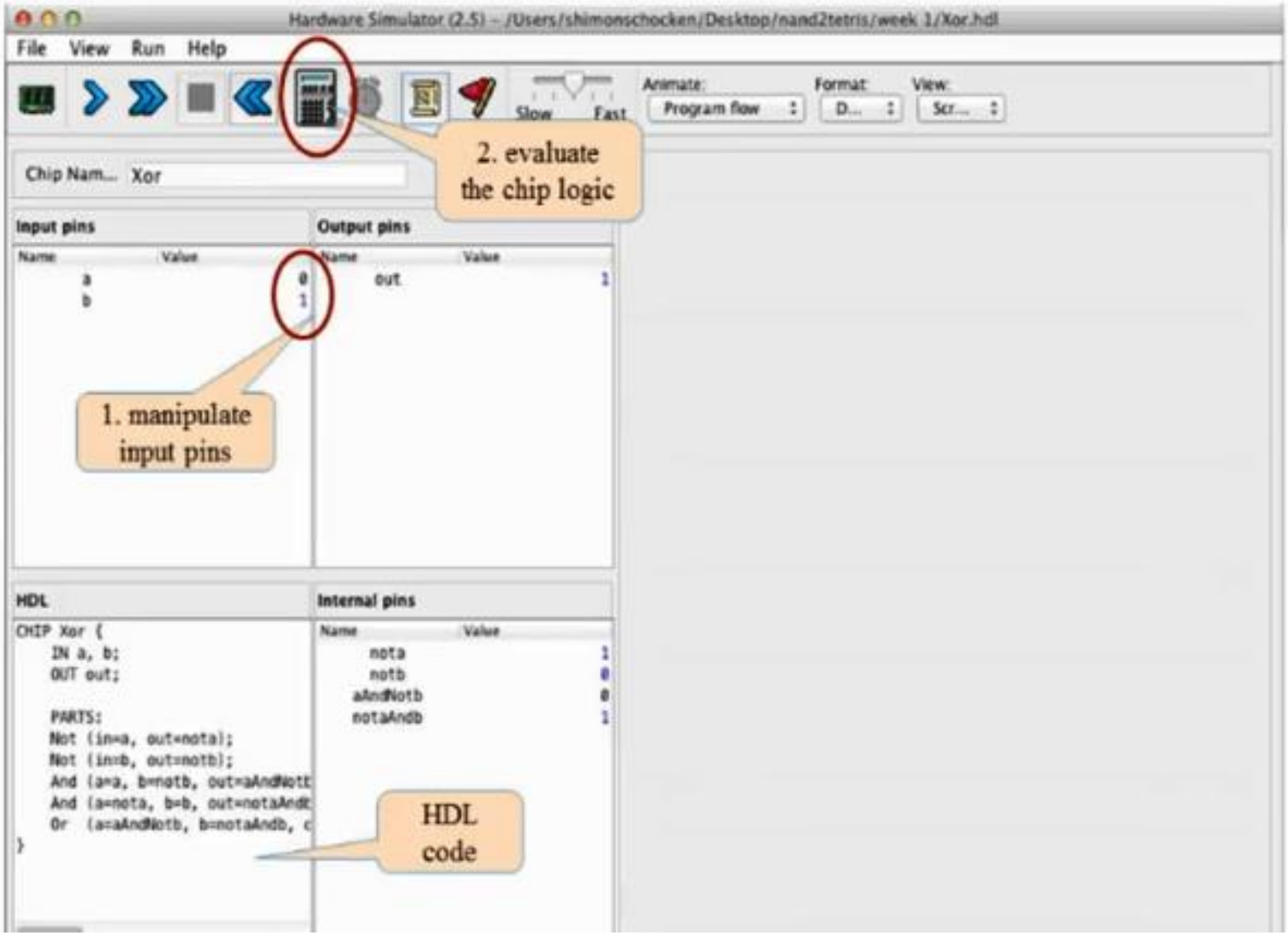
```
CHIP Xor {
    IN a, b;
    OUT out;

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



Simulation process:

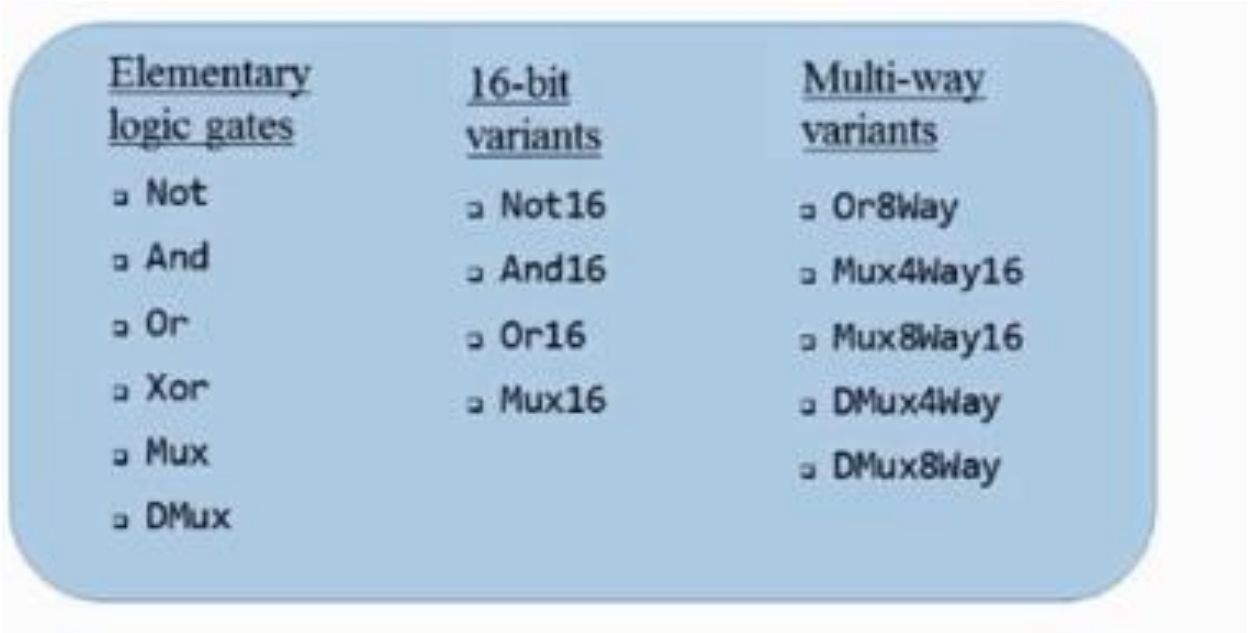
- Load the HDL file into the hardware simulator
- Enter values (0's and 1's) into the chip's input pins (e.g. a and b)
- Evaluate the chip's logic
- Inspect the resulting values of:
  - The output pins (e.g. out)
  - The internal pins (e.g. nota, notb, aAndNotb, notaAndb)

Interactive simulation



Main Parts of CPU

- A basic 16 bit computer processing unit has fallowing parts:

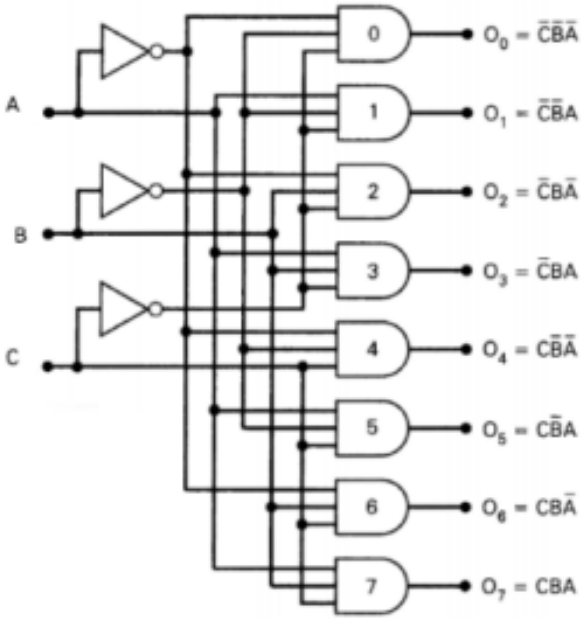


Implement Function via Basic Gates

- In a **decoder** when we give 0,0,0 to the input the first output becomes 1 the others become 0; when we give 0,0,1 to the input the first output becomes 1 the others become 1; it goes like this...
- Boolean Expressions for outputs:
- O7=ABC; O6=ABC'; O5=AB'C; O4=AB'C'; O3=A'BC; O2=A'BC'; O1=A'B'C; O0=A'B'C'
- The truth table and chip design is below

3 to 8 Decoder – Truth Table

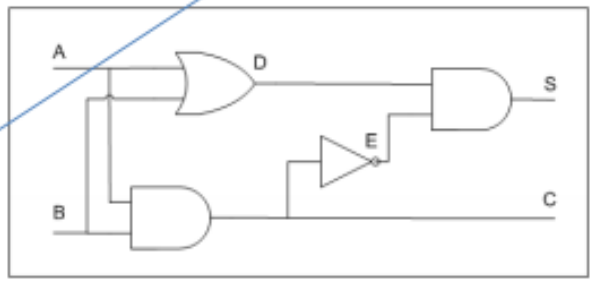
A	B	C	O_7	O_6	O_5	O_4	O_3	O_2	O_1	O_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0



2 bit adder

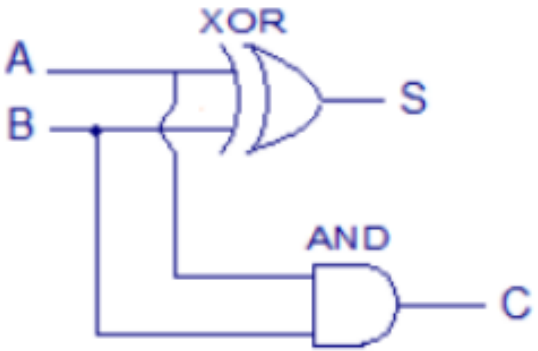
- The boolean expressions of outputs S=A'B+B'A =AxorB ; C=AB

Truth Table			
Input		Output	
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



?

Schematic



Realization

2-bit adder with nand gates

- S=A'B+AB' =AXORB ; C=AB
- we can represent all boolean expressions using NAND

NOT(x)= (x NAND x);

(x AND y)= NOT(x NAND y);

(x OR y)= NOT(x) NAND NOT(y)

A'B=NOT(A' NAND B)

AB'=NOT(A NAND B')

A'B +AB'=NOT NOT(A' NAND B) NAND NOT NOT(A NAND B')

A'B +AB'=(A' NAND B) NAND(A NAND B')

★S=((A NAND A) NAND B) NAND (A NAND (B NAND B))

★C=AB =NOT(A NAND B)=(A NAND B) NAND (A NAND B)

S=((A NAND A) NAND B) NAND (A NAND (B NAND B))

C=(A NAND B) NAND (A NAND B)

★ Circuit Design with NAND:

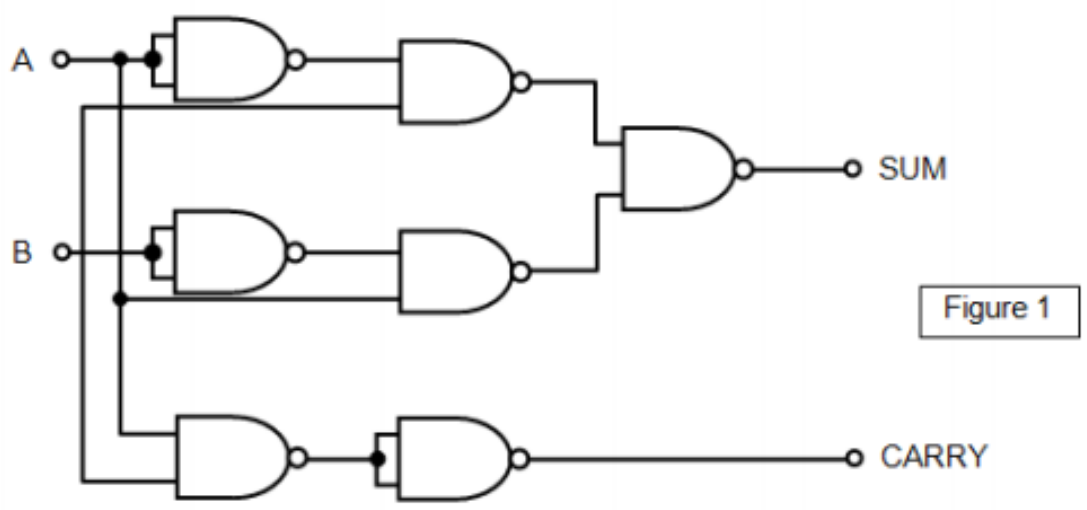


Figure 1