# The Course

**This course is based on the Nand2Tetris course**

“Elements of Computing Systems”, Nisan & Schocken

You will build a complete computer system, layer by layer, starting with Nand gates and ending with a compiler.

**How it will work**

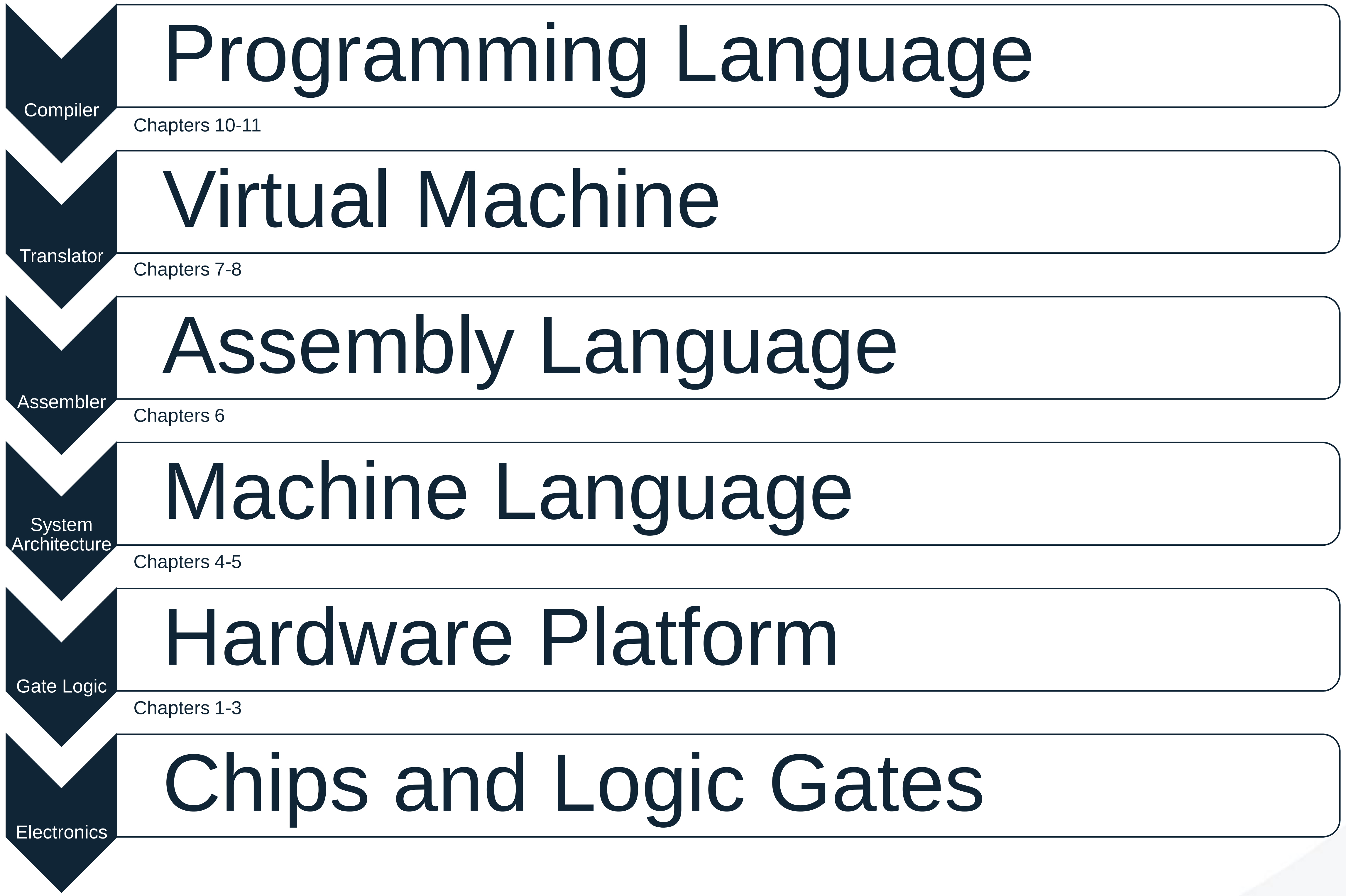
We cover material in lectures supported by the text book, you get to practice in workshops and assignments with quizzes and exams to help you test your knowledge.

Our assignments are not the Nand2Tetris projects.

You will keep a logbook of your software development process that will be used by us to track how you’re going. It will be interleaved with your **svn** commits and web submissions.

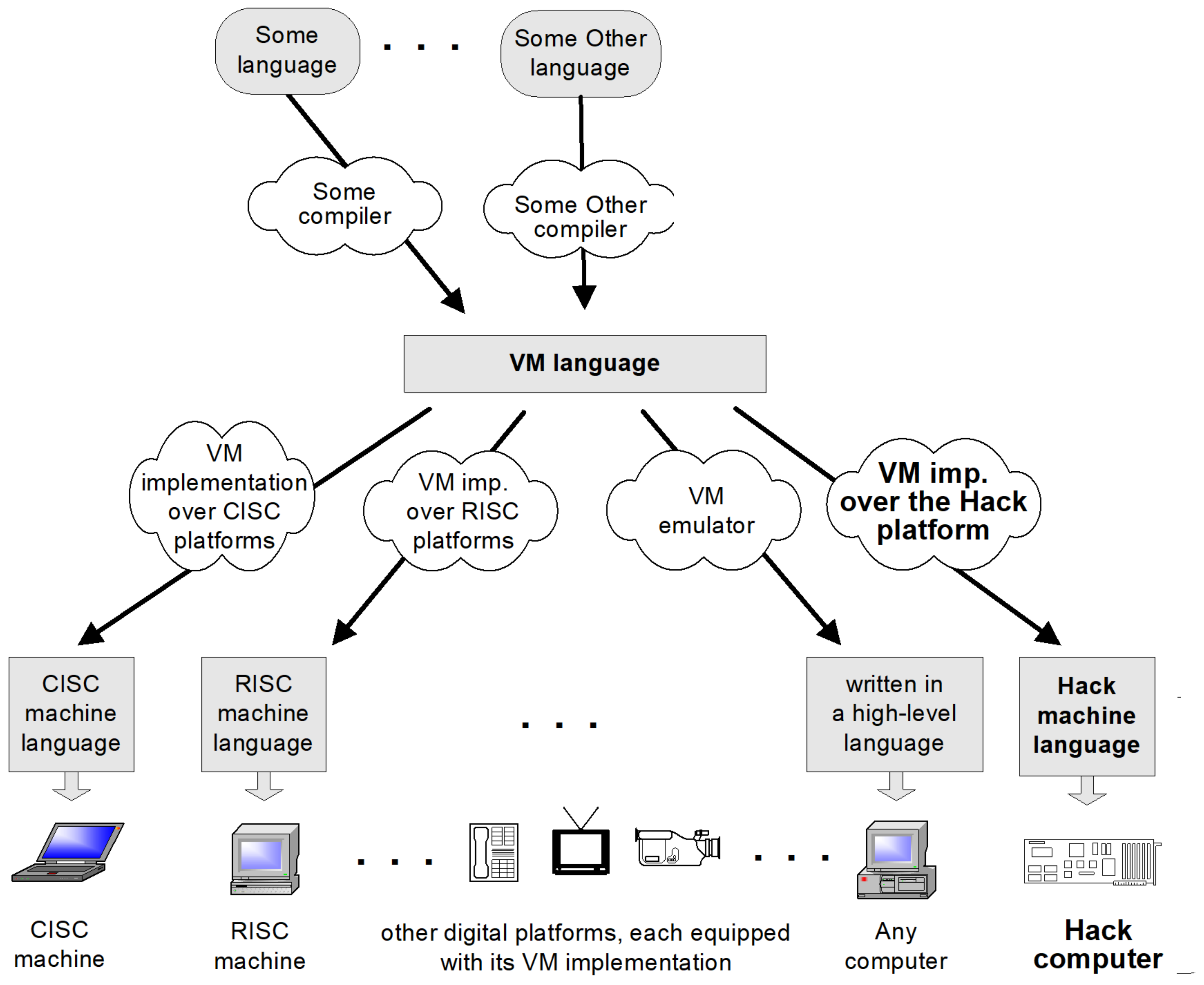
Do all of the work and do it yourself. We are teaching you **effective** techniques at the moment – the goal is your learning rather than just getting projects done!

# The whole system

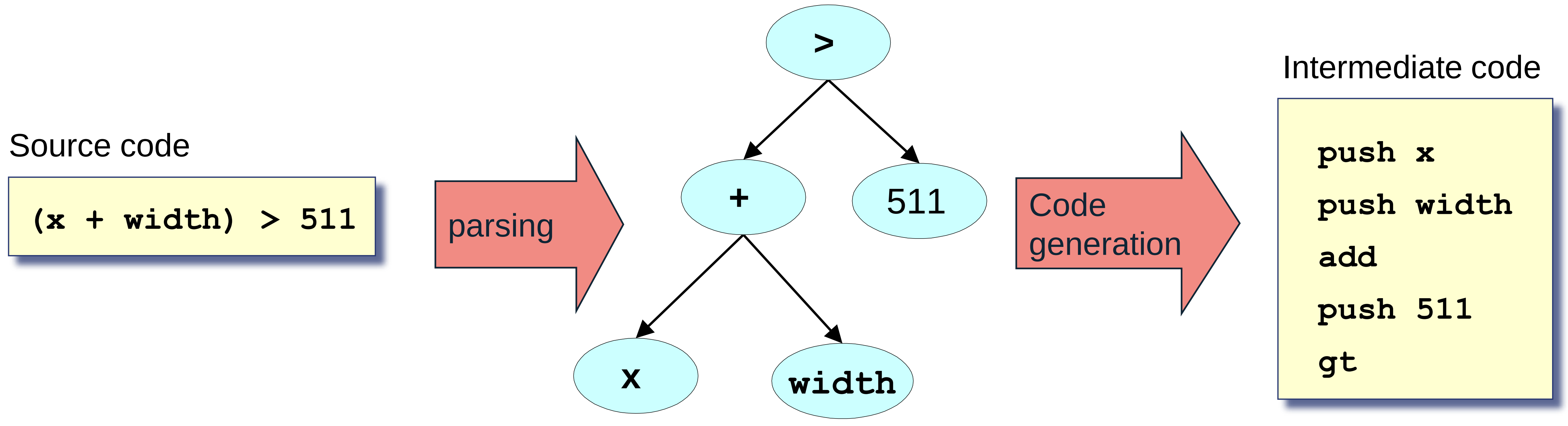


THE UNIVERSITY *of* ADELAIDE

# A modern compilation model



# Compilation



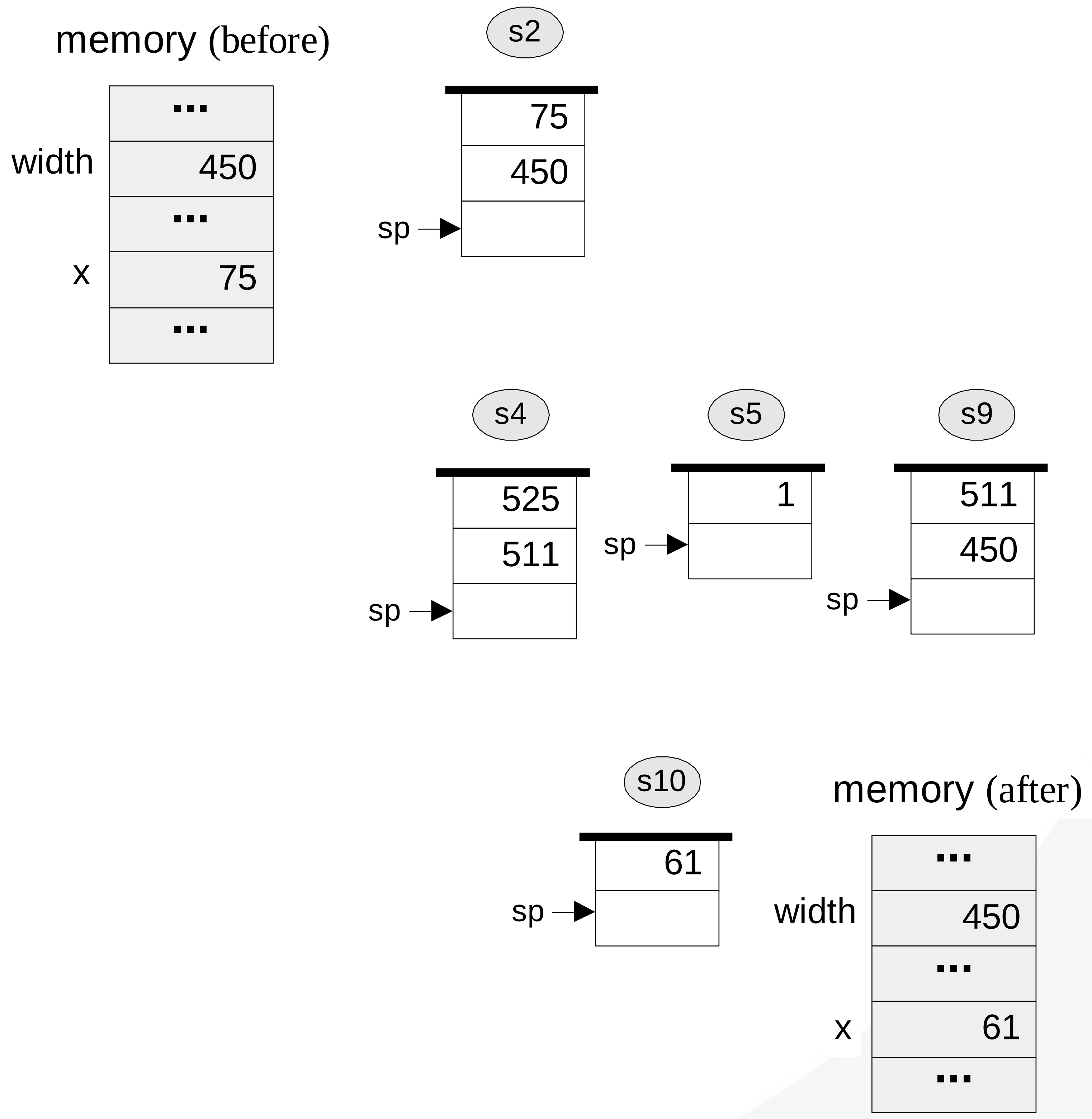
```
push x
push width
add
push 511
gt
```

Abstraction Syntax Analysis Parse Tree Semantic Synthesis

# Inside a virtual machine

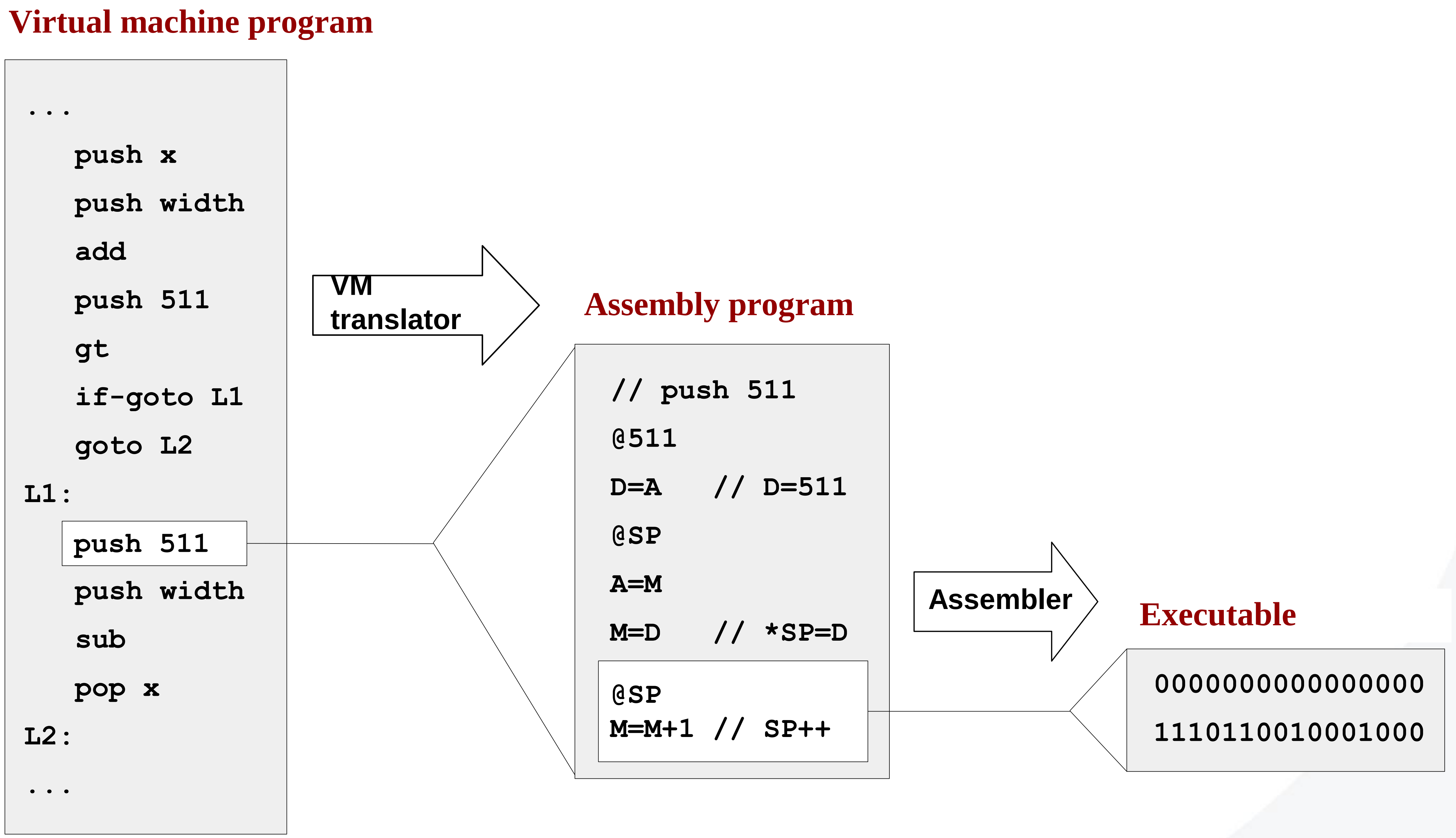
```
if ((x+width)>511)
{
   let x=511-width;
}
```

```
// VM implementation
   push x         // s1
   push width     // s2
   add            // s3
   push 511       // s4
   gt             // s5
   if-goto L1     // s6
   goto L2        // s7
L1:
   push 511       // s8
   push width     // s9
   sub            // s10
   pop x          // s11
L2:
...
```



# The low-level programming path

For now, ignore all details!



Virtual machine program

```
...
   push x
   push width
   add
   push 511
   gt
   if-goto L1
   goto L2
L1:
   push 511
   push width
   sub
   pop x
L2:
...
```

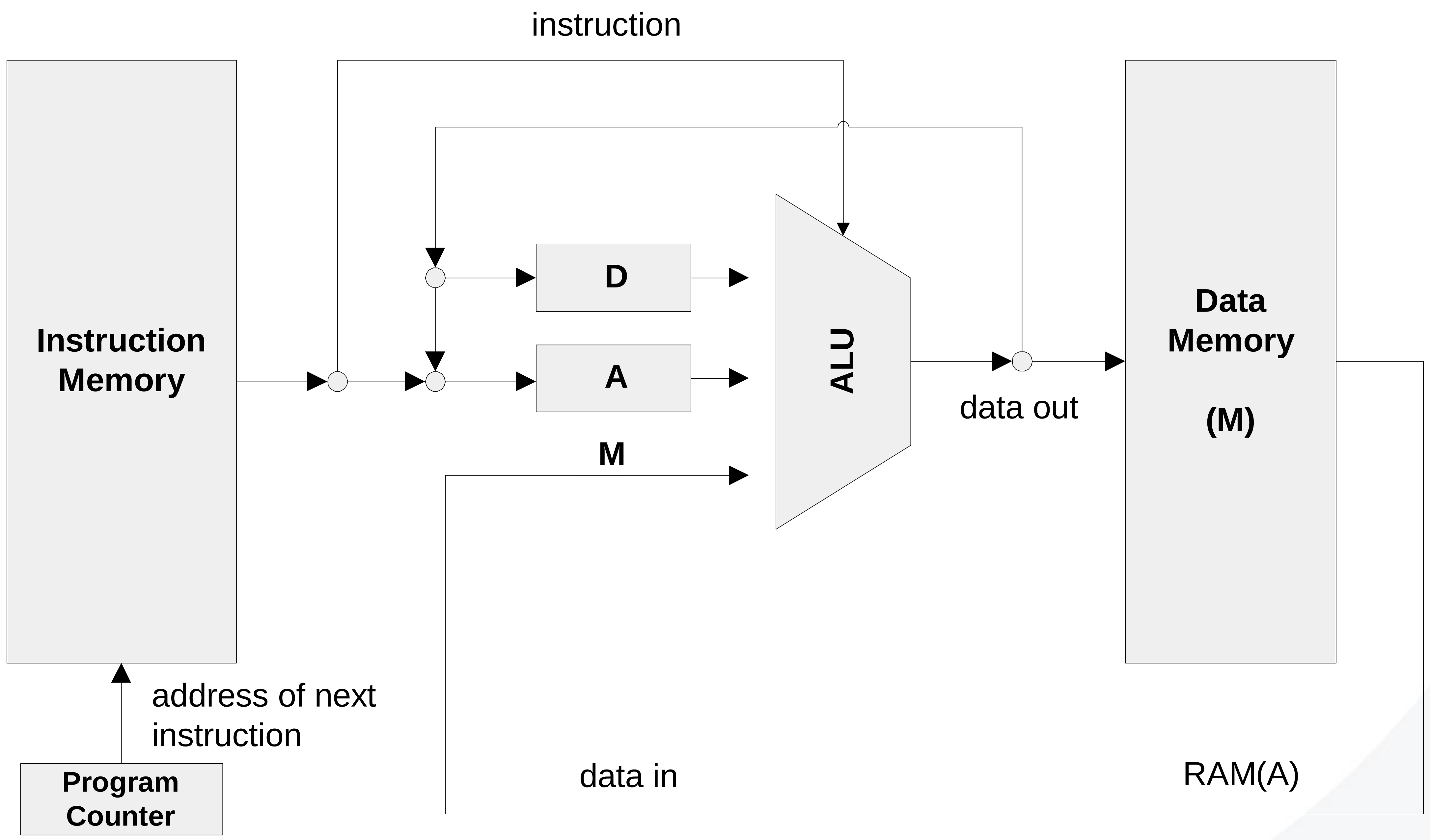
Assembly program

```
// push 511
@511
D=A    // D=511
@SP
A=M
M=D    // *SP=D
@SP
M=M+1 // SP++
```

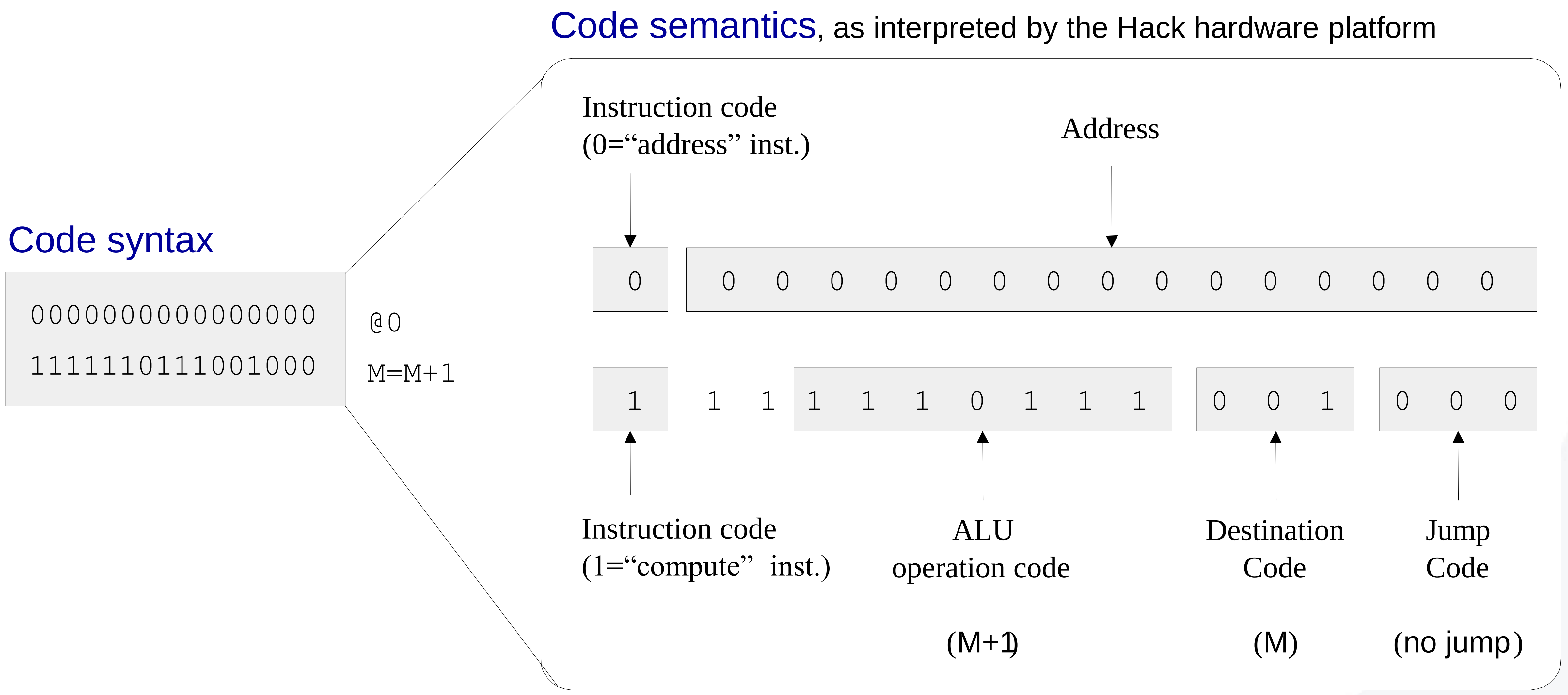
Executable

```
0000000000000000
1110110010001000
```

## What do the instructions do?



# The code directs elements of the processor in order to achieve results



## Summary

We’re going to show you how software and hardware work together to build a computer system.

Over the course, you will build parts of that system and get practice in combinational, sequential and gate logic, as well as learning how high level languages make things happen in real systems.

You have a workshop this week on the tools you’ll need for the course.

After this lecture read the first chapter of the text book.

## Logic design

**Three types of logic we will be using:**

Combinational logic – used for the ALU

Sequential logic – used for RAM

Gate logic – putting it all together to get a computer

## What is gate logic?

**Our hardware is an inter-connected set of chips.**

**Chips are built of simpler chips, down to the simplest structure of all – the elementary logic gate.**

**Logic gates are hardware implementations of Boolean functions. This allows us to represent logical statements in computer form.**

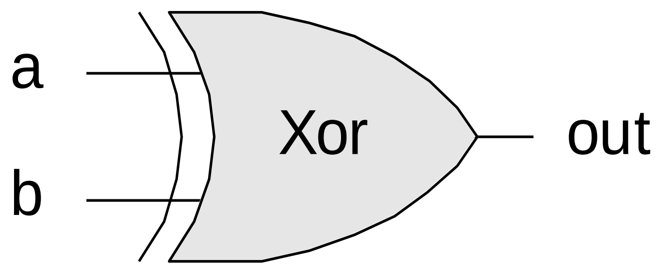
**Every chip and gate has:**

An interface: Telling us what it does

An implementation: Telling us how it does it.

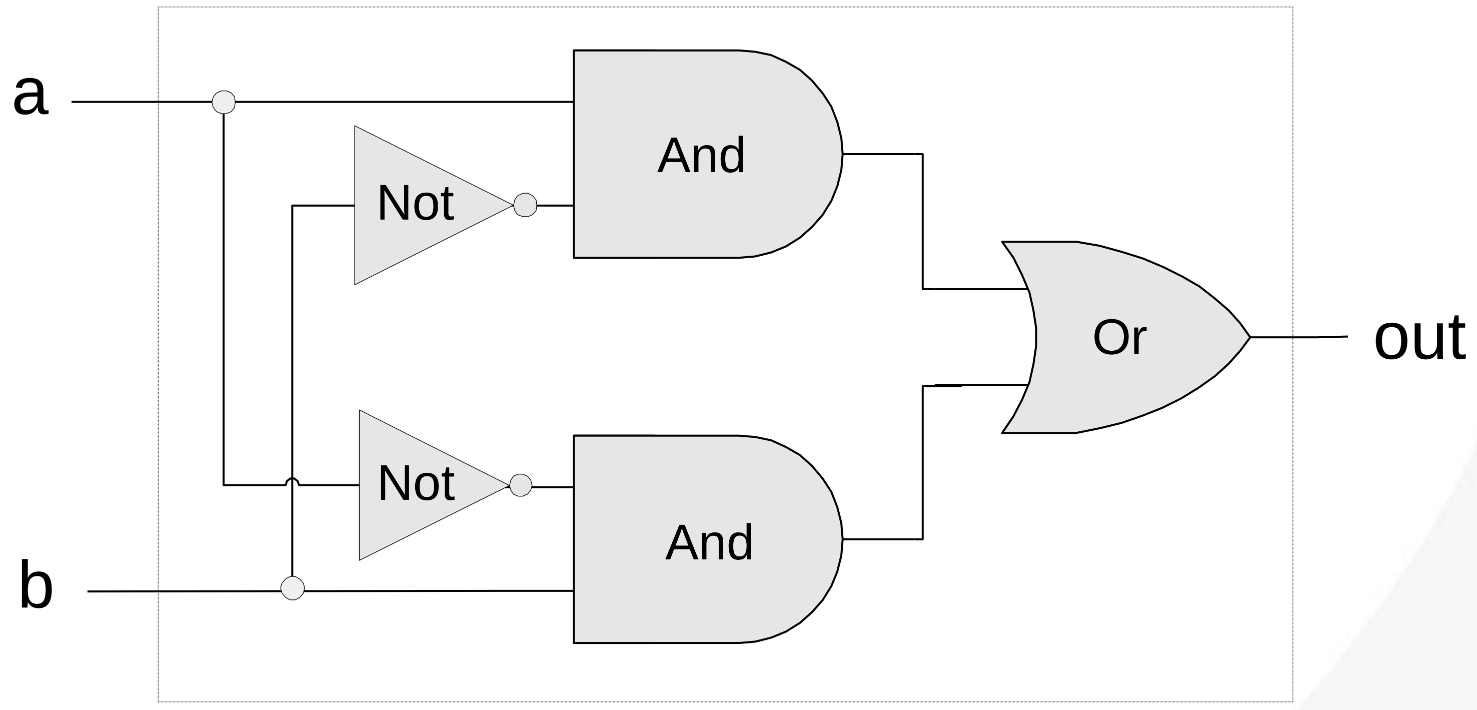
## Example

Interface



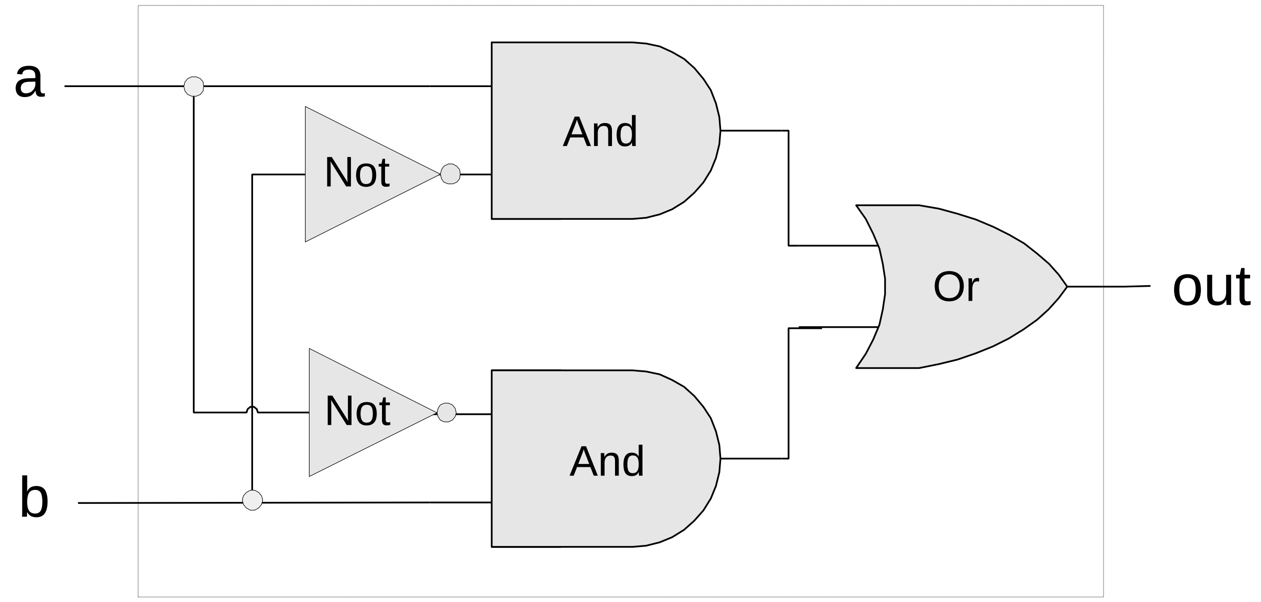
| a | b | out |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Implementation



## Building gates

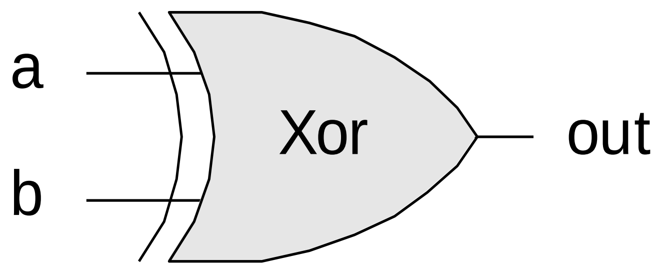
**We won’t be building real gates, we’ll build them in simulation using a Hardware Description Language (HDL)**



```
CHIP Xor {
   IN a,b;
   OUT out;
   PARTS:
   Not(in=a,out=Nota);
   Not(in=b,out=Notb);
   And(a=a,b=Notb,out=w1);
   And(a=Nota,b=b,out=w2);
   Or(a=w1,b=w2,out=out);
}
```

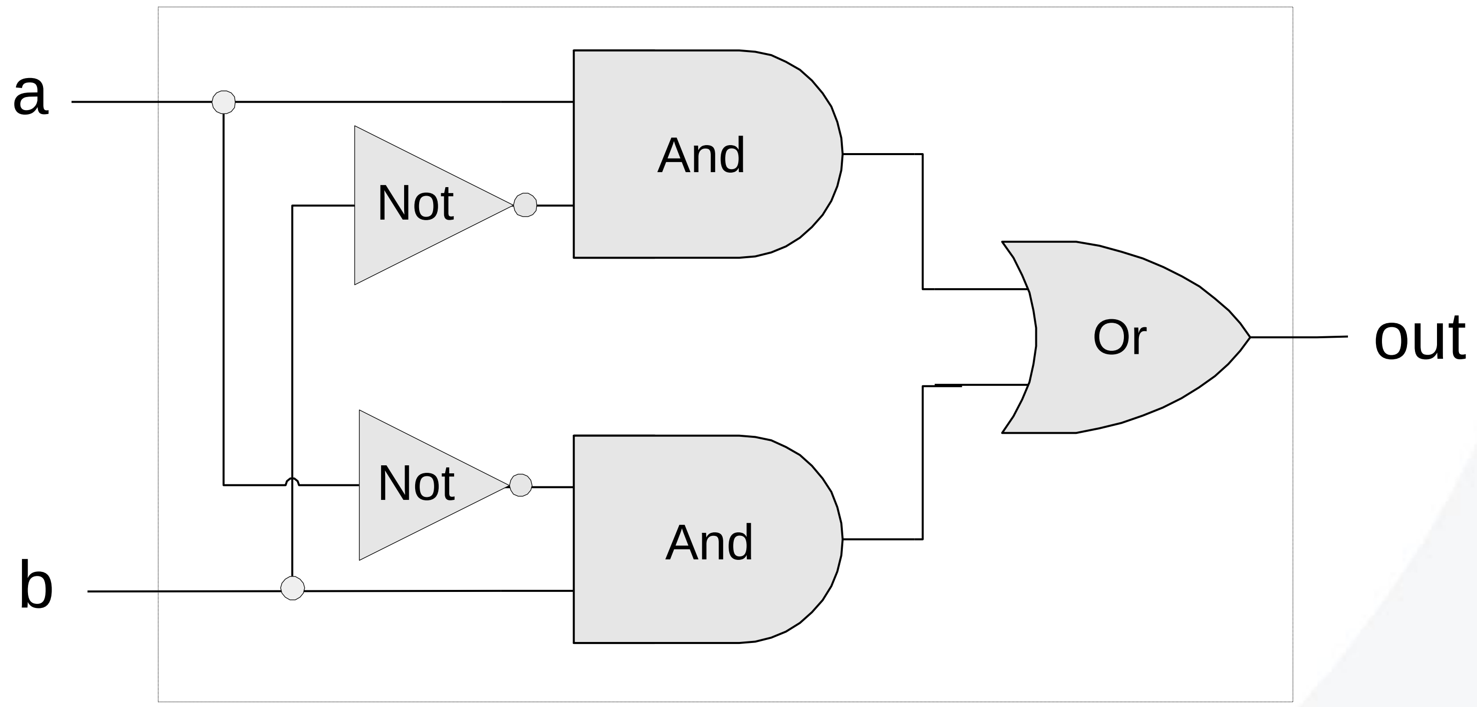
## Review: Example

Interface

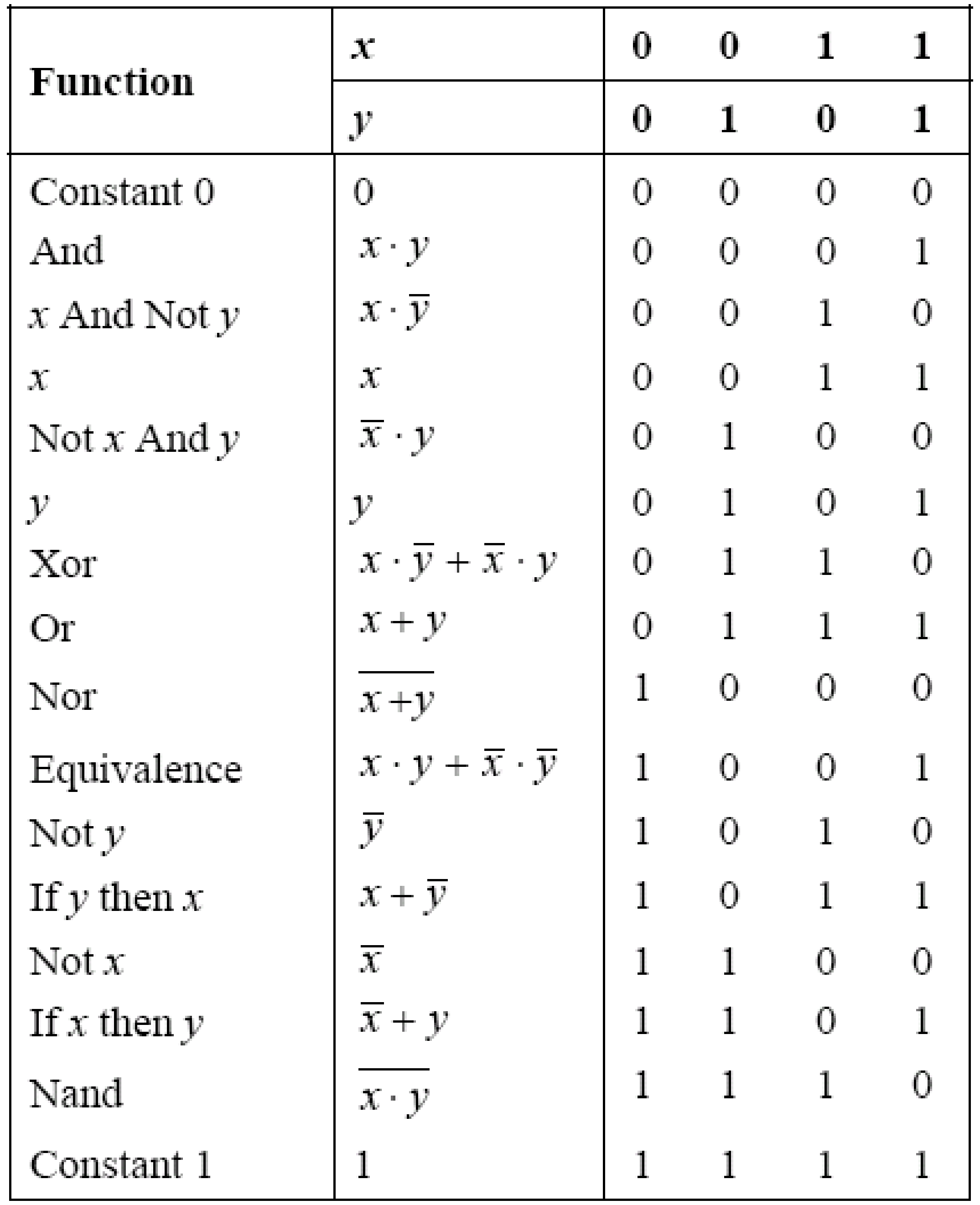


| a | b | out |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Implementation



# All Boolean functions of 2 variables



| | Function | $x$ | 0 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|
| | | $y$ | 0 | 1 | 0 | 1 |
| Constant 0 = $x.\bar{x}$ | Constant 0 | 0 | 0 | 0 | 0 | 0 |
| | And | $x \cdot y$ | 0 | 0 | 0 | 1 |
| | $x$ And Not $y$ | $x \cdot \bar{y}$ | 0 | 0 | 1 | 0 |
| | $x$ | $x$ | 0 | 0 | 1 | 1 |
| | Not $x$ And $y$ | $\bar{x} \cdot y$ | 0 | 1 | 0 | 0 |
| | $y$ | $y$ | 0 | 1 | 0 | 1 |
| Add / Difference | Xor | $x \cdot \bar{y} + \bar{x} \cdot y$ | 0 | 1 | 1 | 0 |
| | Or | $x + y$ | 0 | 1 | 1 | 1 |
| | Nor | $\overline{x+y}$ | 1 | 0 | 0 | 0 |
| XNor | Equivalence | $x \cdot y + \bar{x} \cdot \bar{y}$ | 1 | 0 | 0 | 1 |
| | Not $y$ | $\bar{y}$ | 1 | 0 | 1 | 0 |
| x Or Not y | If $y$ then $x$ | $x + \bar{y}$ | 1 | 0 | 1 | 1 |
| | Not $x$ | $\bar{x}$ | 1 | 1 | 0 | 0 |
| Not x Or y | If $x$ then $y$ | $\bar{x} + y$ | 1 | 1 | 0 | 1 |
| | Nand | $\overline{x \cdot y}$ | 1 | 1 | 1 | 0 |
| Constant 1 = $x+\bar{x}$ | Constant 1 | 1 | 1 | 1 | 1 | 1 |

## Canonical Form

**We can construct a canonical representation of any boolean function**

For each row that gives a 1 in its truth table

- **and** together all terms after applying **not** to any 0 to make it a 1
- if applied to any other row, the equation will evaluate to 0

Then

- **or** together the equations for every row that gives a 1

**XOR**

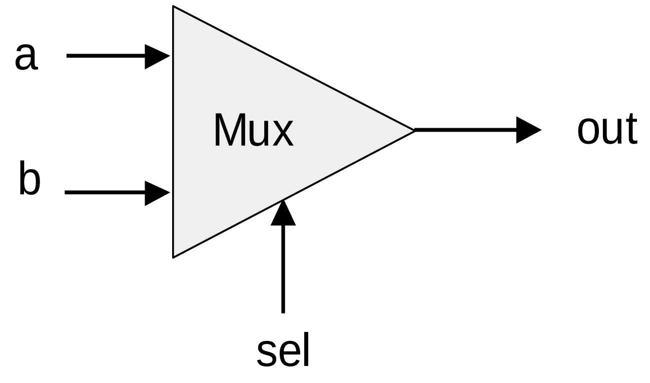
$\overline{x}.y + x.\overline{y}$

| x | y | x^y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**So you only need and, or and not gates**

## Canonical Form – Mux

| a | b | sel | out |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |



```
out = if sel == 0
then a else b
```

# How to construct and, or and not

**From the truth table:**

$\overline{x} = \overline{x.x}$

$x.y = \overline{\overline{x.y}}$

$x+y = \overline{\overline{x}.\overline{y}}$

**Not(x)**

Nand(x,x)

**And(x,y)**

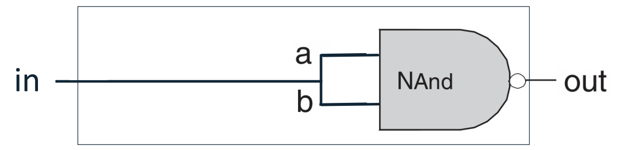
Nand(Nand(x,y),Nand(x,y))

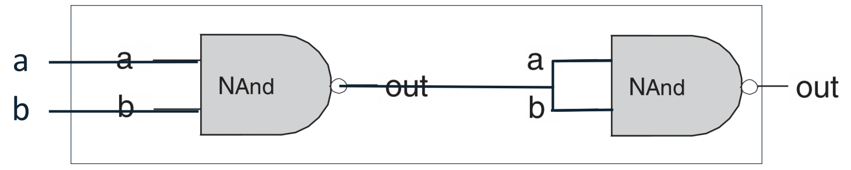
**Or(x,y)**

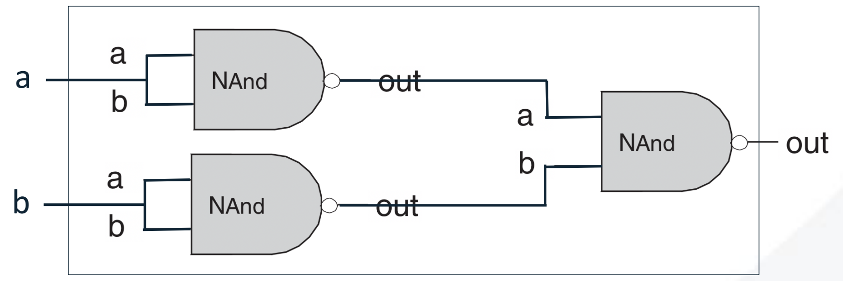
Nand(Not(x),Not(y))

**We only need nand gates!**

| $x$ | $y$ | $x.y$ | $x+y$ | $\overline{x}$ | $\overline{x.y}$ | $\overline{x.x}$ | $\overline{\overline{x.y}}$ | $\overline{\overline{x}.\overline{y}}$ |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |



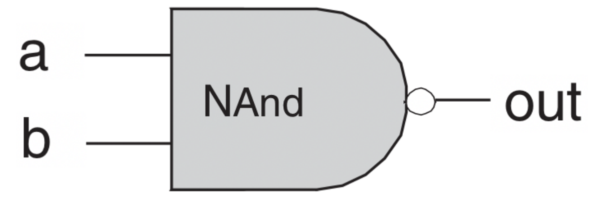




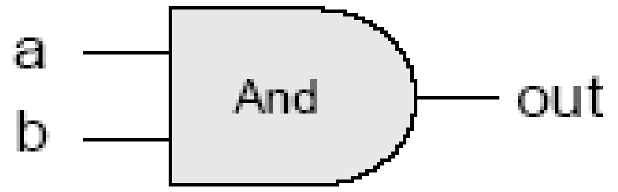
## Summary: Gate logic

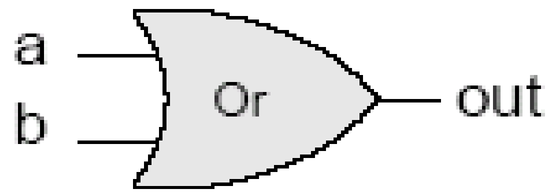
**Gate logic – a gate architecture designed to implement a Boolean function**

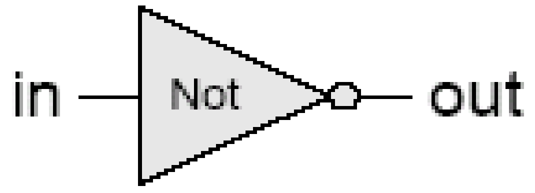
- Elementary gates:

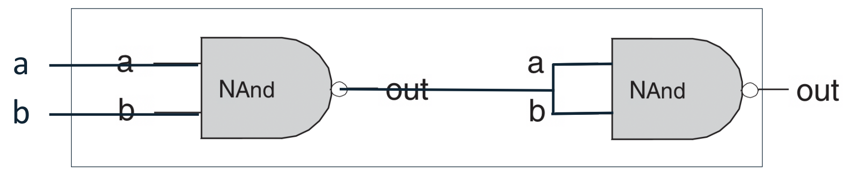


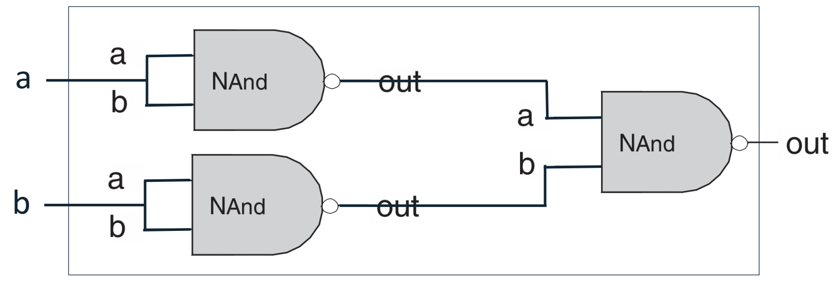
- Composite gates:

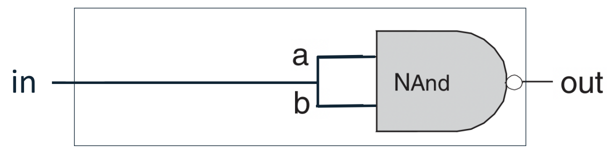






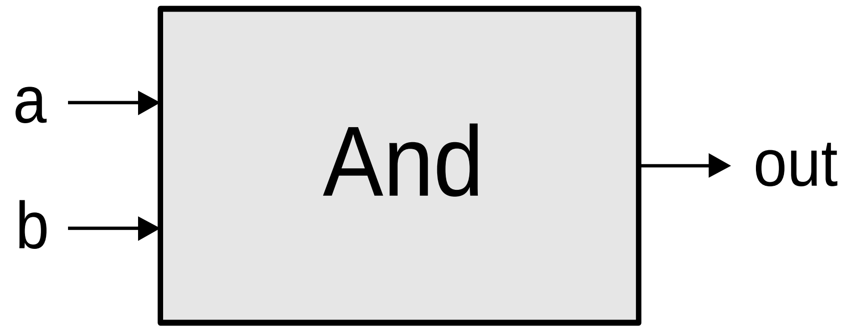






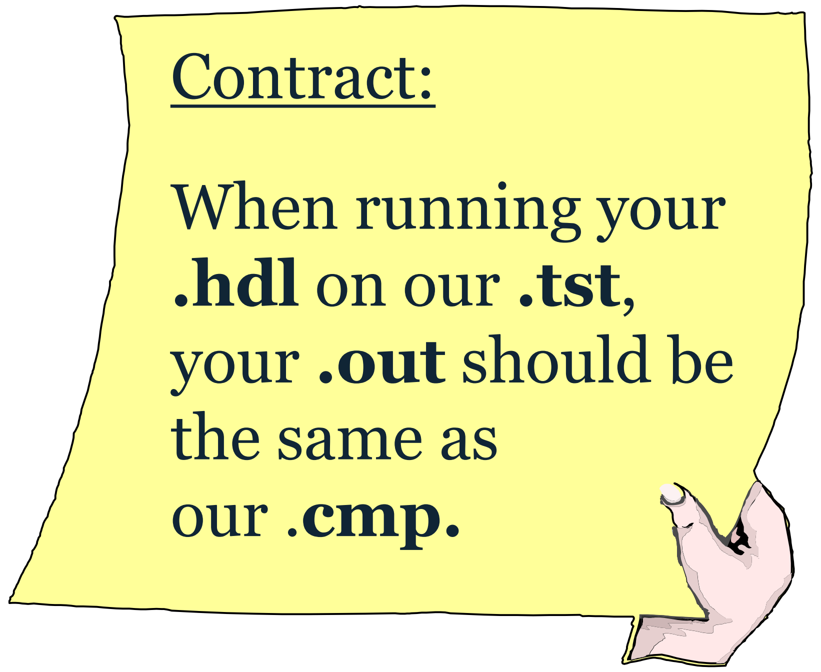
- Important distinction: Interface (*what*) VS implementation (*how*).

# Example: Building an `And` gate



And.cmp

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



Contract:

When running your **.hdl** on our **.tst**, your **.out** should be the same as our **.cmp.**

And.hdl

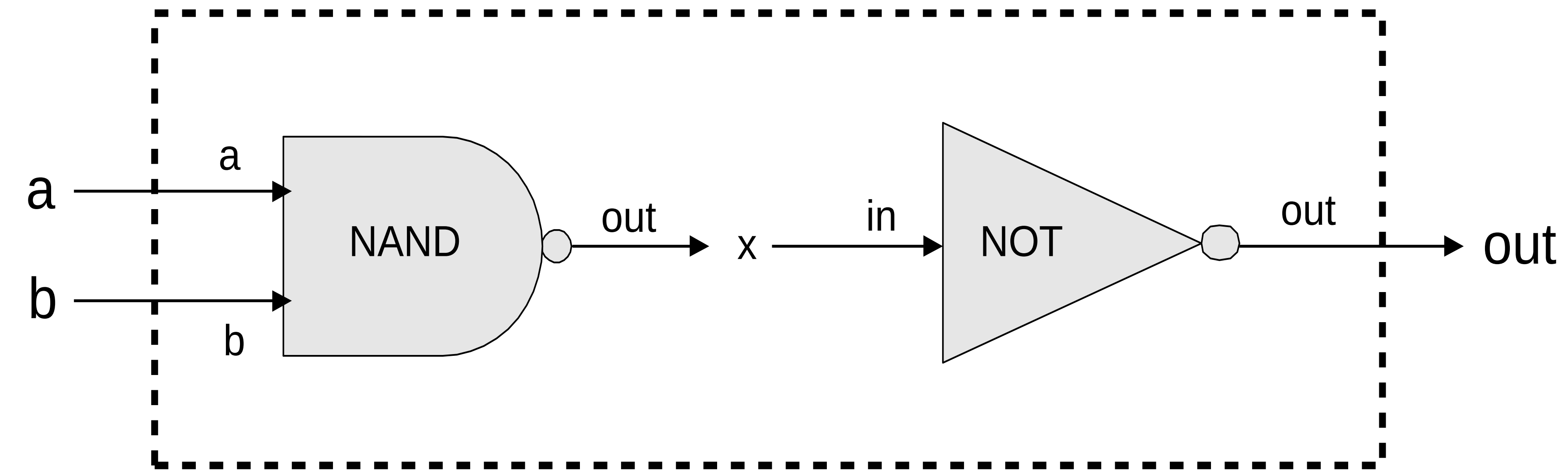
```
CHIP And
{   IN  a, b;
    OUT out;

    PARTS:
    // implementation missing
}
```

And.tst

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

# Building an `And` gate

Implementation: And(a,b) = Not(Nand(a,b))

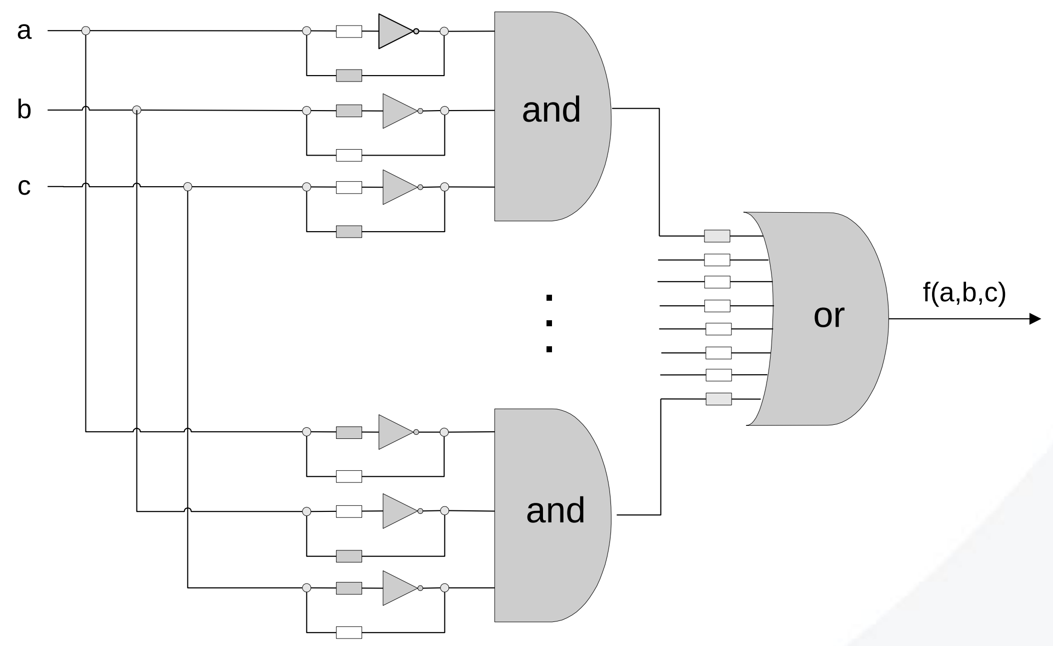


And.hdl `And(a = ?, b = ?, out = ?);`

```
CHIP And
{    IN  a, b;
     OUT out;
     Nand(a = a, b = b, out = x);
     Not(in = x, out = out);
     // Nand(a = x, b = x, out = out);
}
```

# Boolean Functions!

- Each Boolean function has a canonical representation
- The canonical representation is expressed in terms of And, Not, Or
- And, Not, Or can be expressed in terms of Nand alone (or Nor)
- Every Boolean function can be realized by a standard circuit consisting of Nand gates only
- Mass production
- Universal building blocks, unique topology



## This Week

- Review Chapter 1 of the Text Book
- Start Assignment 1 (available Wednesday)
- Workshops start this week