Course Overview

Shimon Schocken

Spring 2005

Course goals

Course objectives:

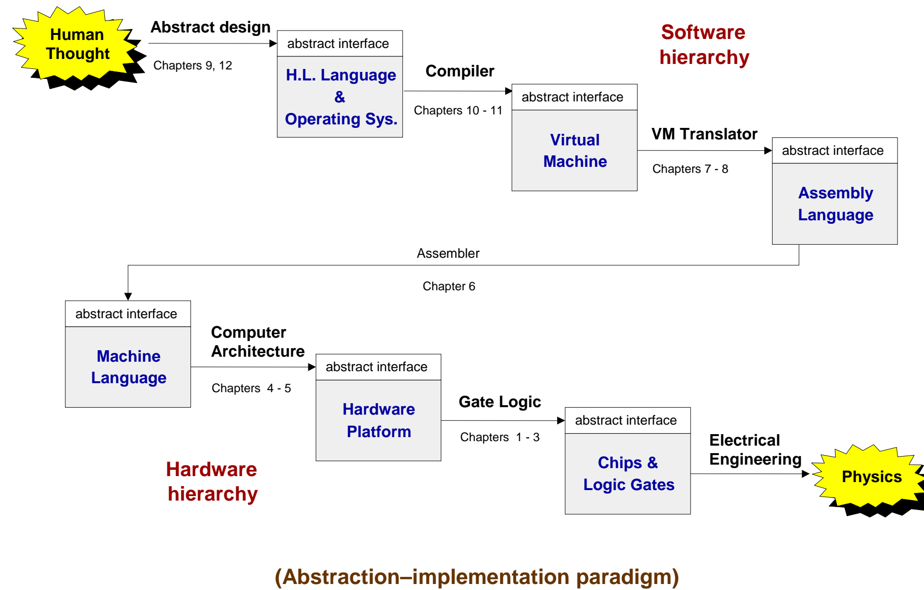
- Understand how HW+SW systems are built, and how they work
- Learn how to break complex problems into simpler ones
- Learn how large scale development projects are planned and executed
- Have fun.

Methodology:

- Build / experiment with a transparent computer that we can fully understand.

Copyright © Shimon Schocken

Course theme and structure



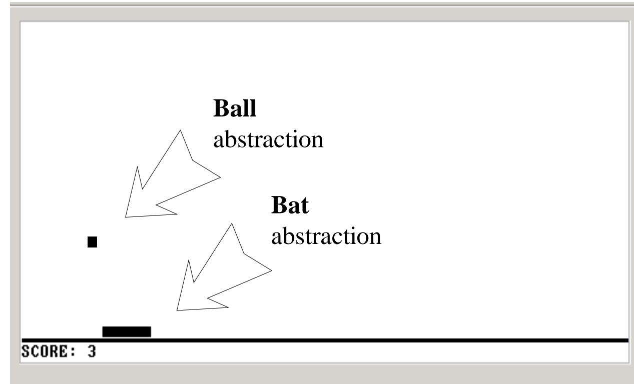
(Abstraction–implementation paradigm)

Resources and rules

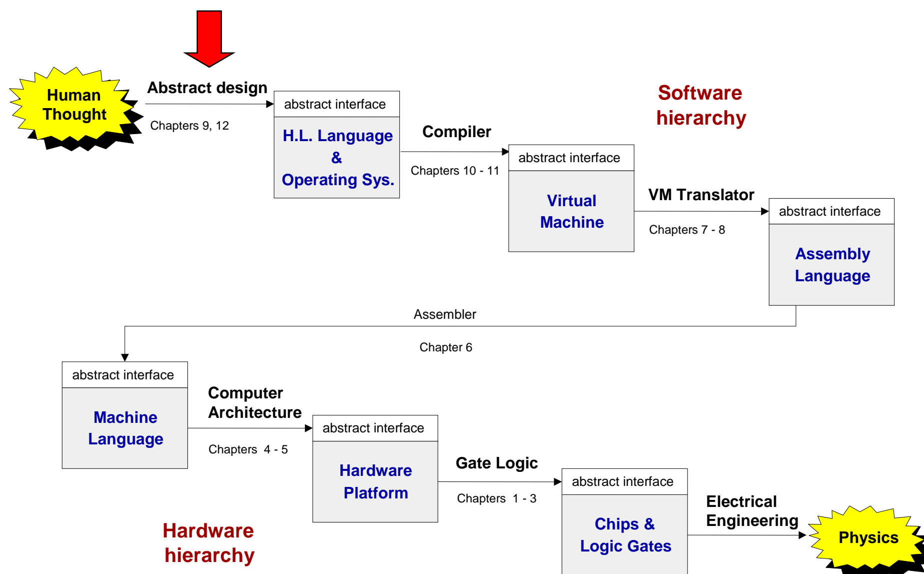
- Book
- Lectures
- Exercises
- Tools
- Course site
- Exam
- Projects
- Individual work policy

Copyright © Shimon Schocken

Application level: Pong



The big picture



Copyright © Shimon Schocken

High-level programming

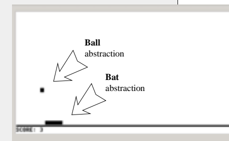
```
/** A Graphic Bat for a Pong Game */
class Bat {
    field int x, y;             // screen location of the bat's top-left corner
    field int width, height;    // bat's width & height

    // The class constructor and most of the class methods are omitted

    /** Draws (color=true) or erases (color=false) the bat */
    method void draw(boolean color) {
       do Screen.setColor(color);
       do Screen.drawRectangle(x,y,x+width,y+height);
       return;
    }

    /** Moves the bat one step (4 pixels) to the right. */
    method void moveR() {
       do draw(false);  // erase the bat at the current location
       let x = x + 4;   // change the bat's X-location
       // but don't go beyond the screen's right border
        if ((x + width) > 511) {
           let x = 511 - width;
       }
       do draw(true);  // re-draw the bat in the new location
       return;
    }
}
```

A typical call to an operating system method



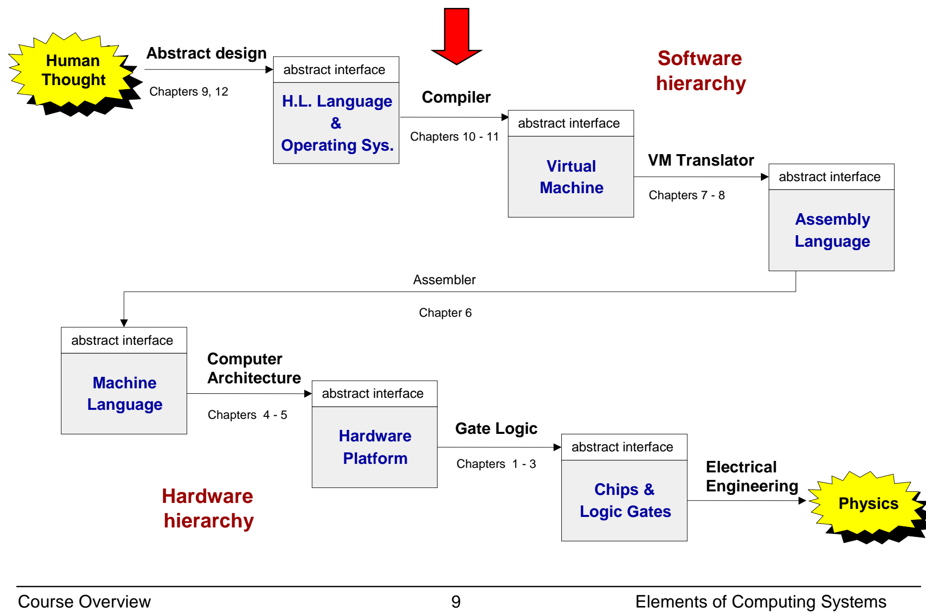
Operating system level

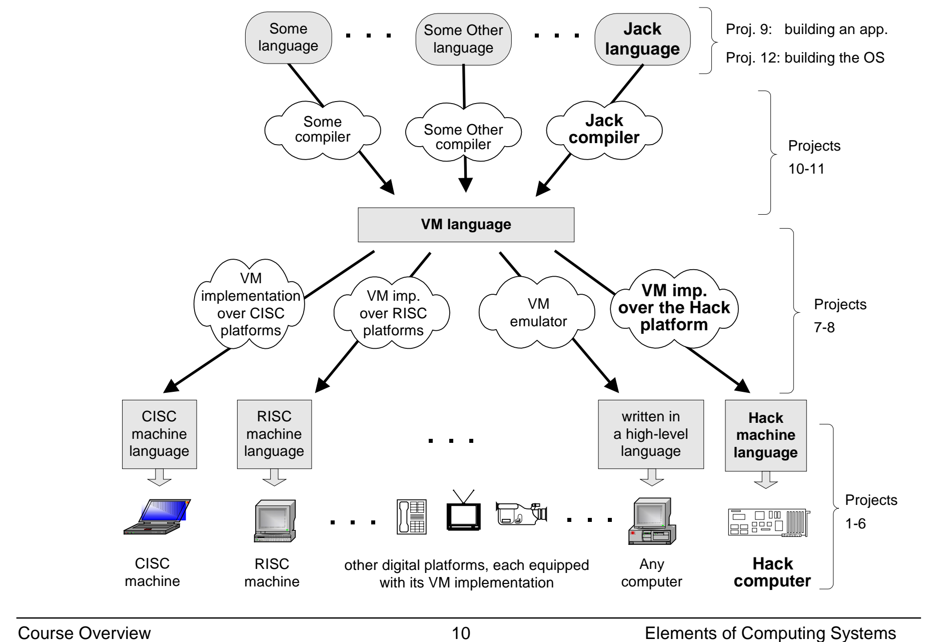
```
/** An OS-level screen driver that abstracts the computer's physical screen */
class Screen {
     static boolean currentColor;  // the current color

     // The Screen class is a collection of methods, each implementing one
     // abstract screen-oriented operation.  Most of this code is omitted.

     /** Draws a rectangle in the current color. */
     // the rectangle's top left corner is anchored at screen location (x0,y0)
     // and its width and length are x1 and y1, respectively.
     function void drawRectangle(int x0, int y0, int x1, int y1) {
         var int x, y;
         let x = x0;
         while (x < x1) {
             let y = y0;
             while(y < y1) {
                do Screen.drawPixel(x,y);
                let y = y+1;
             }
             let x = x+1;
         }
     }
}
```

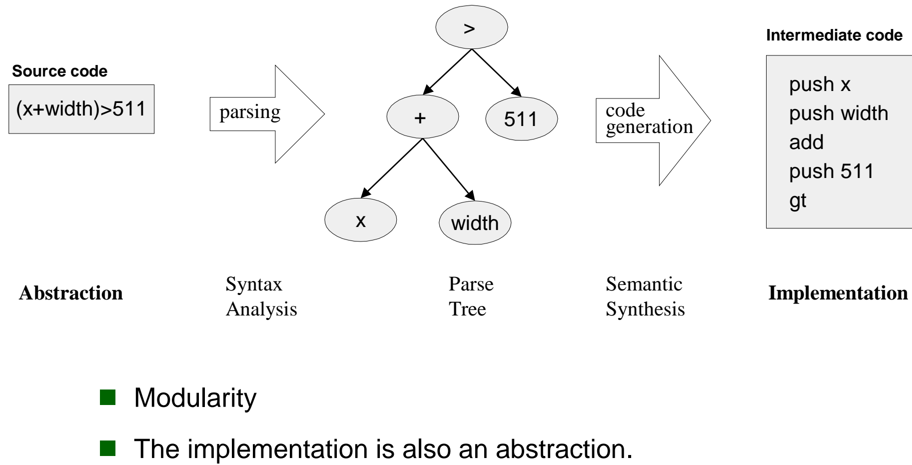

Copyright © Shimon Schocken





Copyright © Shimon Schocken

Compilation



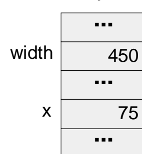
- Modularity
- The implementation is also an abstraction.

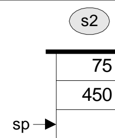
The Virtual Machine (VM)

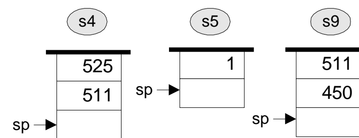
```
if ((x+width)>511) {
   let x=511-width;
}
```

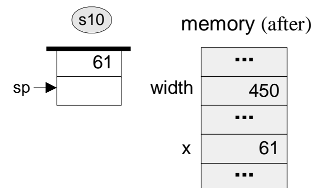
```
// VM implementation
   push x        // s1
   push width    // s2
   add           // s3
   push 511      // s4
   gt            // s5
   if-goto L1    // s6
   goto L2       // s7
L1:
   push 511      // s8
   push width    // s9
   sub           // s10
   pop x         // s11
L2:
...
```

memory (before)



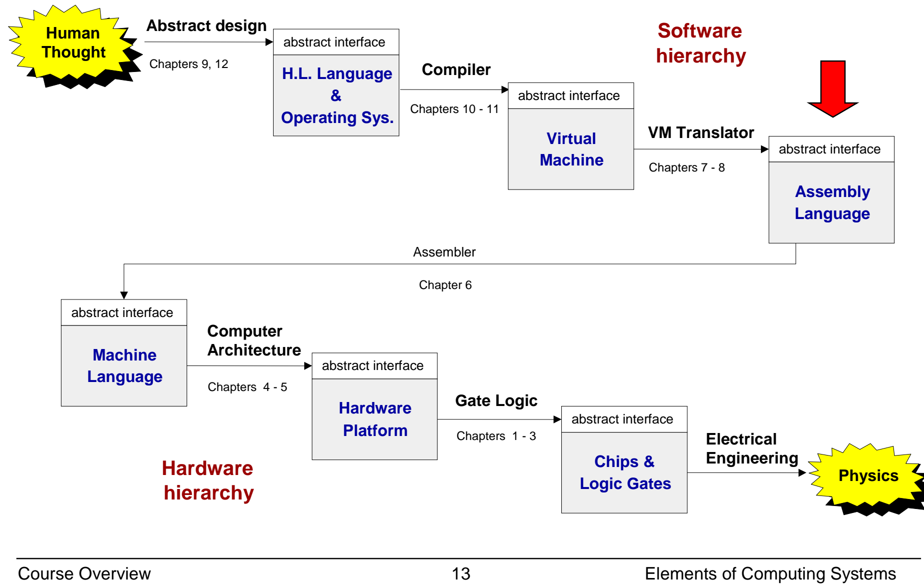






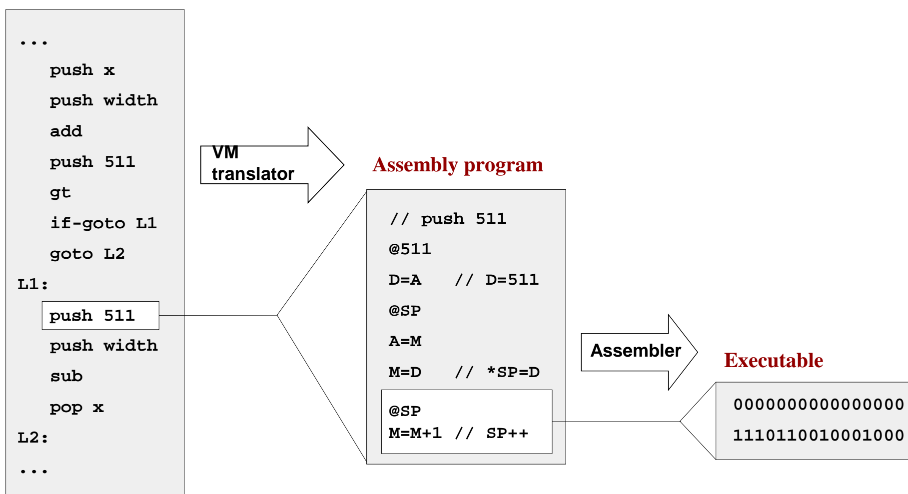
Copyright © Shimon Schocken

The big picture



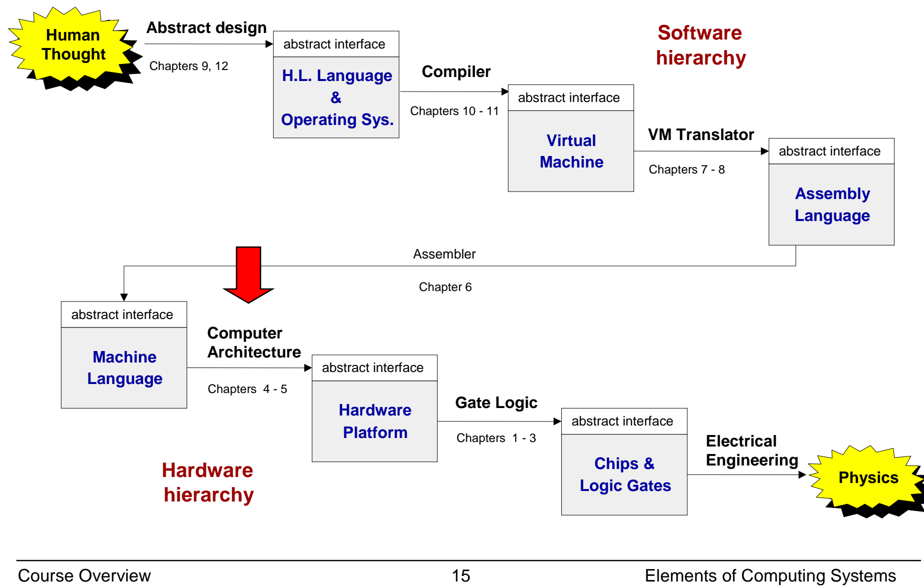
Low-level programming

Virtual machine program

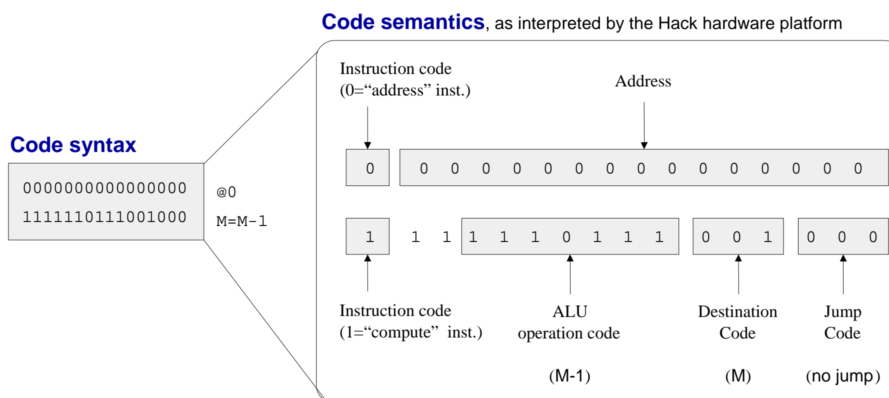


Copyright © Shimon Schocken

The big picture



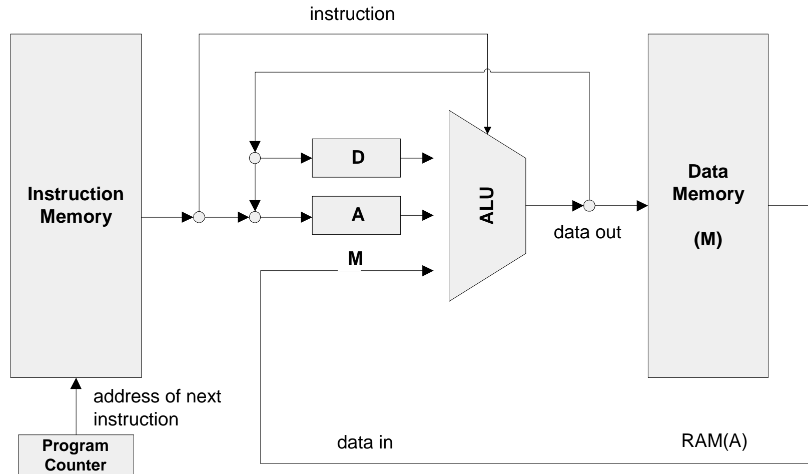
Machine language semantics



- We need a HW architecture that will realize this semantics
- The HW platform should be designed to:
  - Parse instructions, and
  - Execute them

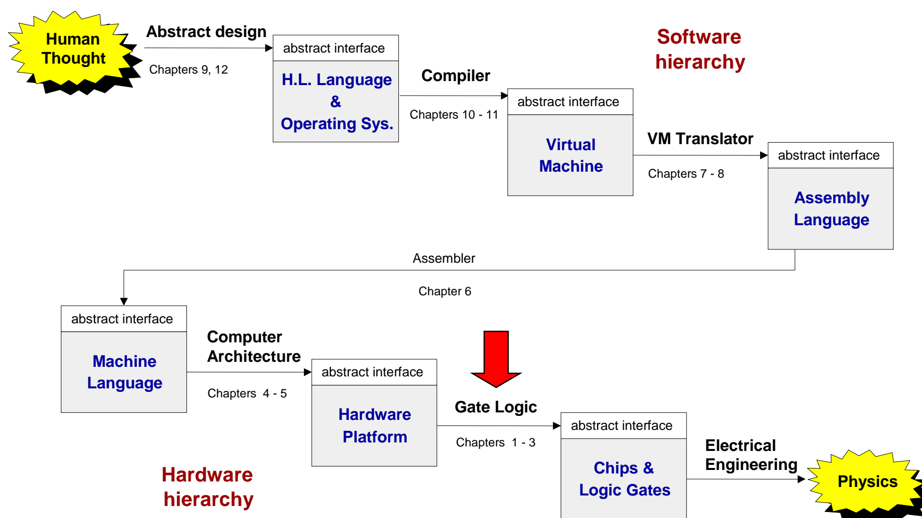
Copyright © Shimon Schocken

Computer architecture



- A typical Von Neumann machine

The big picture

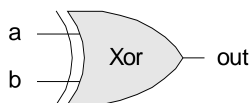


Copyright © Shimon Schocken

Gate logic

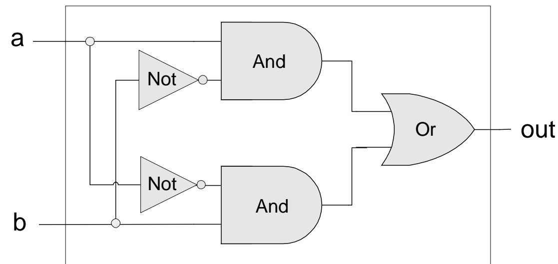
- HW platform = inter-connected set of chips
- Chips are made of simpler chips, all the way down to logic gates
- Logic gate = HW element that implements a certain Boolean function
- Every chip and gate has an interface, specifying WHAT it is doing, and an implementation, specifying HOW it is doing it.

Interface

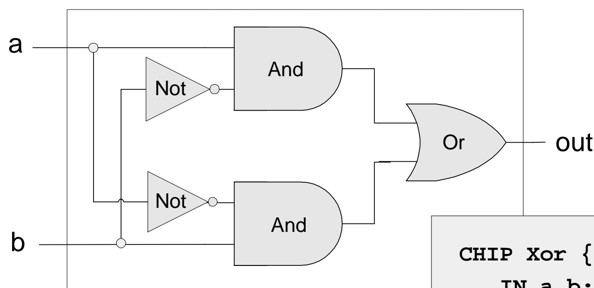


a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

Implementation



Hardware Description Language (HDL)

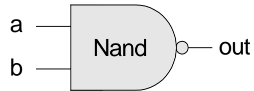



```
CHIP Xor {
   IN a,b;
   OUT out;
   PARTS:
   Not(in=a,out=Nota);
   Not(in=b,out=Notb);
   And(a=a,b=Notb,out=w1);
   And(a=Nota,b=b,out=w2);
   Or(a=w1,b=w2,out=out);
}
```


Copyright © Shimon Schocken

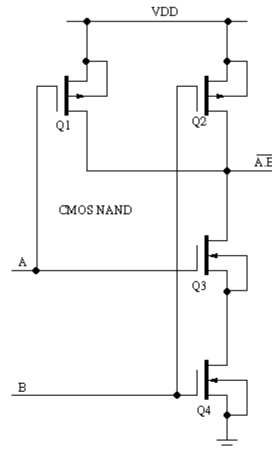
The tour ends

Interface



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

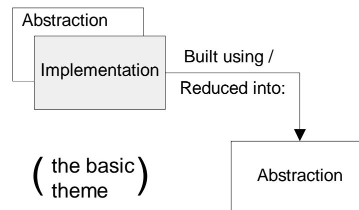
Implementation (CMOS)



On the power of abstractions

<u>Abstraction:</u> an attempt to capture in thought the essence of something

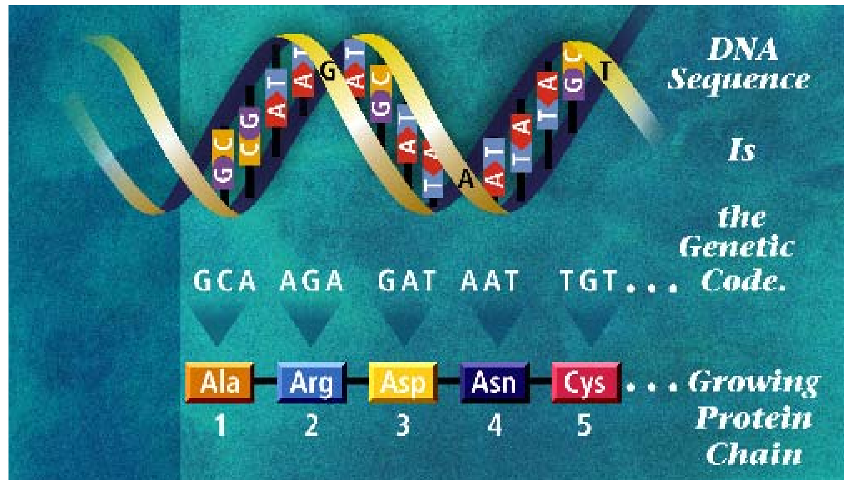
- Cognitive
- Math
- Sciences
- Computer science
- Top down / bottom up



(the basic theme)

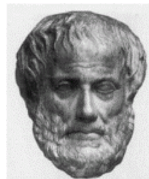
Copyright © Shimon Schocken

Famous abstraction



Final note

We deliberate not about ends, but about means. For a doctor does not deliberate whether he shall heal, nor an orator whether he shall persuade ... They assume the end and consider *how* and by what means it is attained, and if it seems easily and best produced thereby; And if it is achieved by some means, they consider how it will be achieved, and by what means *this* will be achieved, until they come to the first cause. And what is last in the order of analysis seems to be first in the order of becoming.



Copyright © Shimon Schocken

Boolean Logic

Shimon Schocken

Boolean algebra

Some elementary Boolean operators:

- Not(x)
- And(x,y)
- Or(x,y)
- Nand(x,y)

x	Not(x)
0	1
1	0

x	y	And(x,y)
0	0	0
0	1	0
1	0	0
1	1	1

x	y	Or(x,y)
0	0	0
0	1	1
1	0	1
1	1	1

x	y	Nand(x,x)
0	0	1
0	1	1
1	0	1
1	1	0

Boolean functions:

x	y	z	$f(x,y,z)=(x+y)\overline{z}$
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

- Functional expression VS truth table expression
- Important result: Every Boolean function can be expressed using And, Or, Not

Copyright © Shimon Schocken

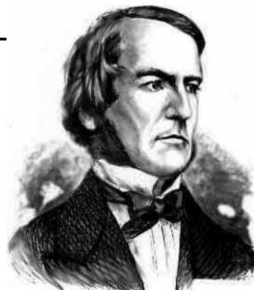
All Boolean functions of 2 variables

Function	x	0	0	1	1
	y	0	1	0	1
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
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
Equivalence	$x \cdot y + \bar{x} \cdot \bar{y}$	1	0	0	1
Not y	$\bar{y}$	1	0	1	0
If y then x	$x + \bar{y}$	1	0	1	1
Not x	$\bar{x}$	1	1	0	0
If x then y	$\bar{x} + y$	1	1	0	1
Nand	$\overline{x \cdot y}$	1	1	1	0
Constant 1	1	1	1	1	1

Boolean algebra

Given: `Nand(a,b), false`

- `Not(a) = Nand(a,a)`
- `true = Not(false)`
- `And(a,b) = ...`
- `Or(a,b) = ...`
- `Xor(a,b) = ...`
- Etc.

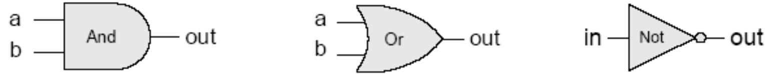


George Boole, 1815-1864
("*A Calculus of Logic*")

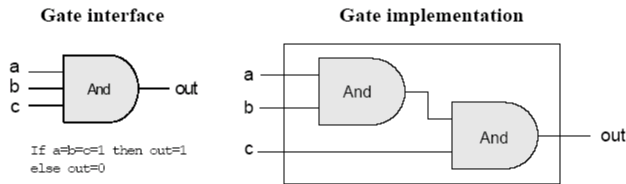
Copyright © Shimon Schocken

Gate logic

- Gate logic – a gate architecture designed to implement a boolean function
- Elementary gates:

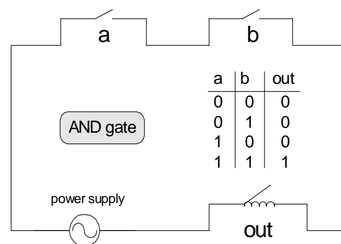


- Composite gates:

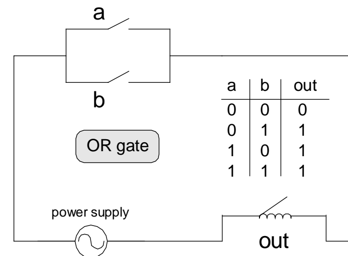


- Interface VS implementation.

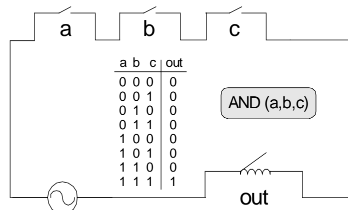
Circuit implementations



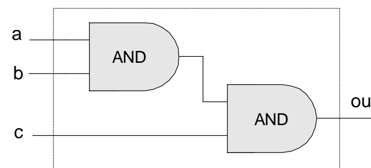
a	b	out
0	0	0
0	1	0
1	0	0
1	1	1



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



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

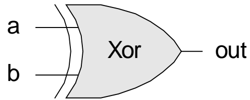


- Physical realizations of logic gates are irrelevant to computer science.

Copyright © Shimon Schocken

Gate Logic

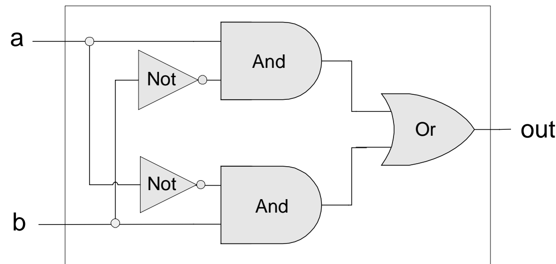
Interface



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

Claude Shannon, 1916-2001
("Symbolic Analysis of Relay and Switching Circuits")

Implementation



Xor(a,b)=Or(a,Not(b),Not(a),b))

Project 1: elementary logic gates

Given: `Nand(a,b), false`

a	b	Nand(a,b)
0	0	1
0	1	1
1	0	1
1	1	0

Build:

- `Not(a) = ...`
- `true = ...`
- `And(a,b) = ...`
- `Or(a,b) = ...`
- `Mux(s,a,b) = ...`
- Etc. - 12 gates altogether.

Copyright © Shimon Schocken

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

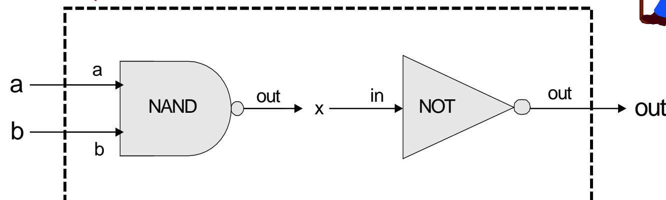
```
CHIP And
{   IN  a, b;
    OUT out;
    // implementation missing
}
```

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

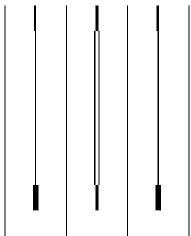
Implementation: And(a,b) = Not(Nand(a,b))



And.hdl

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


Copyright © Shimon Schocken



Hardware simulator

Hardware Simulator - D:\hack\Chips\Project 1\Xor.hdl

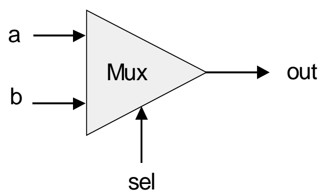
HDL program

```
// Xor (exclusive or) gate
// if a<>b out=1 else out=0
CHIP Xor {
   IN a,b;
   OUT out;
   PARTS:
   Not (in=a,out=nota);
   Not (in=b,out=notb);
   And (a=a,b=notb,out=x);
   And (a=nota,b=b,out=y);
   Or  (a=x,b=y,out=out);
}
```

Script restarted

Multiplexor (an interesting chip)

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



sel	out
0	a
1	b

- Implementation: based on Not, And, Or gates.

Copyright © Shimon Schocken

Canonical representation

Suspect function (a-la-Leibnitz): Each suspect may or may not have an alibi (*a*), a motivation to commit the crime (*m*), and a relationship to the weapon found in the scene of the crime (*w*). The police decides to focus attention only on suspects for whom the proposition Not(*a*) And (*m* Or *w*) is true.

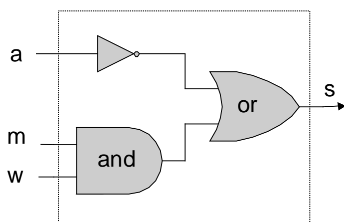
Truth table of the "suspect" function $s(a,m,w) = \bar{a} \cdot (m + w)$

a	m	w	minterm	suspect(a,m,w)= not(a) and (m or w)
0	0	0	$m_0 = \bar{a}\bar{m}\bar{w}$	0
0	0	1	$m_1 = \bar{a}\bar{m}w$	1
0	1	0	$m_2 = \bar{a}m\bar{w}$	1
0	1	1	$m_3 = \bar{a}mw$	1
1	0	0	$m_4 = a\bar{m}\bar{w}$	0
1	0	1	$m_5 = a\bar{m}w$	0
1	1	0	$m_6 = am\bar{w}$	0
1	1	1	$m_7 = amw$	0

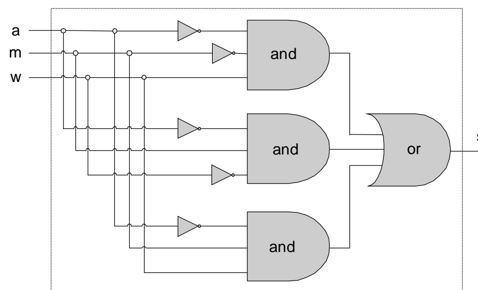
Canonical form: $s(a,m,w) = \bar{a}\bar{m}w + \bar{a}m\bar{w} + \bar{a}mw$

Two possible implementations

$s(a,m,w) = \bar{a} \cdot (m + w)$

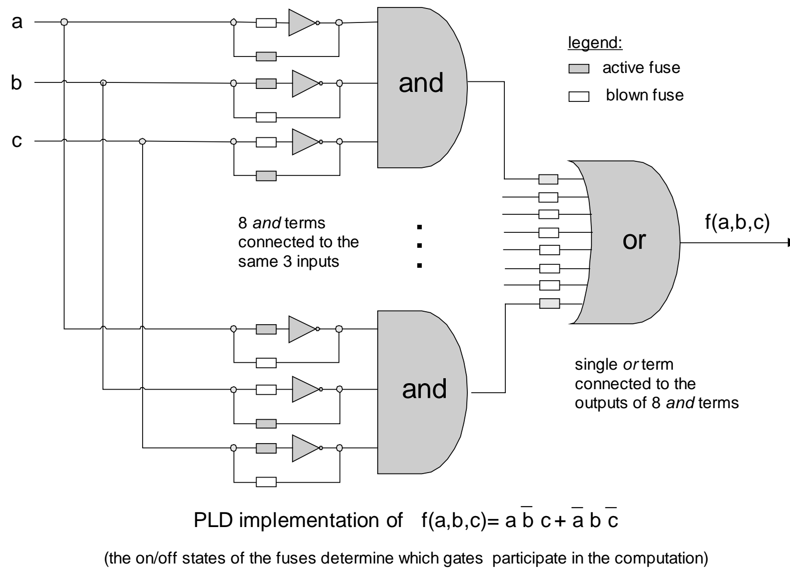


$s(a,m,w) = \bar{a}\bar{m}w + \bar{a}m\bar{w} + \bar{a}mw$



Copyright © Shimon Schocken

Programmable Logic Device for 3-way functions

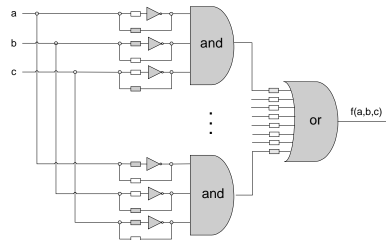


PLD implementation of $f(a,b,c) = a\,\bar{b}\,c + \bar{a}\,b\,\bar{c}$

(the on/off states of the fuses determine which gates participate in the computation)

Some observations

- 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
- Ergo, every Boolean function can be realized by a strandard PLD consisting of Nand gates only
- Mass production
- Universal building blocks, unique connectivity (neurons).



Copyright © Shimon Schocken

Boolean Arithmetic

Shimon Schocken

Counting systems

quantity	decimal	binary	3-bit register
	0	0	000
*	1	1	001
**	2	10	010
***	3	11	011
****	4	100	100
*****	5	101	101
******	6	110	110
*******	7	111	111
********	8	1000	overflow
*********	9	1001	overflow
**********	10	1010	overflow

Copyright © Shimon Schocken

Rationale

$$(9038)_{ten} = 9\cdot 10^3 + 0\cdot 10^2 + 3\cdot 10^1 + 8\cdot 10^0 = 9038$$

$$(10011)_{two} = 1\cdot 2^4 + 0\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 = 19$$

$$(x_n x_{n-1} \ldots x_0)_b = \sum_{i=0}^{n} x_i \cdot b^i$$

Binary addition

- Assuming a 4-bit system:

```
  0 0 0 1          1 1 1 1
    1 0 0 1          1 0 1 1
    0 1 0 1          0 1 1 1
  ---------        ---------
  0 1 1 1 0        1 0 0 1 0

 no overflow        overflow
```

- Algorithm: exactly the same as in decimal addition
- Overflow (MSB carry) has to be dealt with.

Copyright © Shimon Schocken

Representing negative numbers (4-bit system)

0	0000		
1	0001	1111	-1
2	0010	1110	-2
3	0011	1101	-3
4	0100	1100	-4
5	0101	1011	-5
6	0110	1010	-6
7	0111	1001	-7
		1000	-8

- The codes of all positive numbers begin with a "0"
- The codes of all negative numbers begin with a "1"
- To convert a number: leave all trailing 0's and first 1 intact, and flip all the remaining bits

Example: 2 - 5 = 2 + (-5) =

```
  0 0 1 0
+ 1 0 1 1
---------
  1 1 0 1   =  -3
```

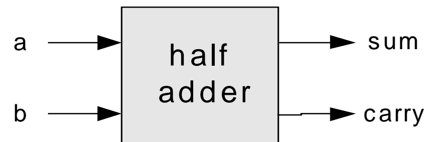

Building an Adder chip

- Adder: a chip designed to add two integers
- The construction hierarchy:
  - Half adder: designed to add 2 bits
  - Full adder: designed to add 3 bits
  - Adder: designed to add two *n*-bit numbers

Copyright © Shimon Schocken

Half adder (designed to add 2 bits)

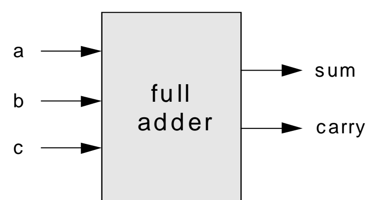
a	b	carry	sum
0	0	0	0
0	0	0	1
0	1	0	1
0	1	1	0



- Implementation: based on two gates that you've seen before.

Full adder (designed to add 3 bits)

a	b	c	carry	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



- Implementation: can be based on half-adder gates.

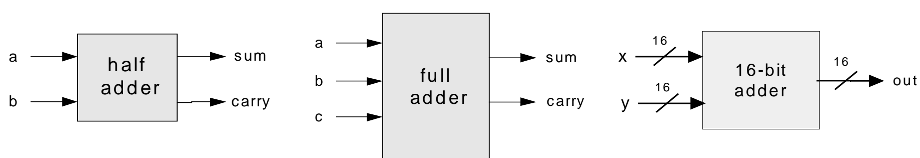
Copyright © Shimon Schocken

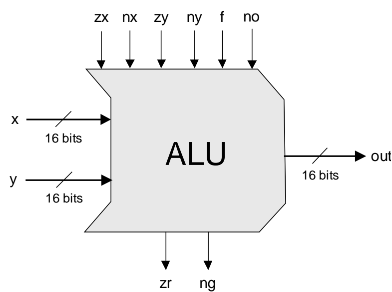
n-bit Adder

...	1	0	1	1	a
					+
...	0	0	1	0	b
...	1	1	0	1	out

- Implementation: array of full-adder gates

The ALU (of the Hack platform)






```
out(x, y, control bits) =
  x+y, x-y, y-x,
  0, 1, -1,
  x, y, -x, -y,
  x!, y!,
  x+1, y+1, x-1, y-1,
  x&y, x|y
```

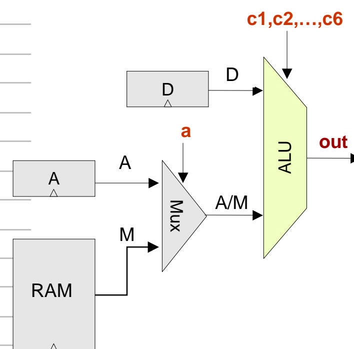

Copyright © Shimon Schocken

ALU logic

These bits instruct how to pre-set the x input		These bits instruct how to pre-set the y input		This bit selects between + / And	This bit inst. how to post-set out	Resulting ALU output
zx	nx	zy	ny	f	no	out=
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x And y	if no then out=!out	f(x,y)=
1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x|y

The ALU in the CPU context

out (when a=0)	c1	c2	c3	c4	c5	c6	out (when a=1)
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D|A	0	1	0	1	0	1	D|M



Copyright © Shimon Schocken

End note: Leibnitz

- Described a binary calculus and a 4-bit adder in 1694
- "The binary system may be used in place of the decimal system; express all numbers by unity and by nothing"
- Practice: built one of the first mechanical calculators
- Theory: dreamed about a univeral, formal, language of thought -- the "Characteristica Universalis"
- The dream's end: Turing and Goedl in 1930's.

Copyright © Shimon Schocken