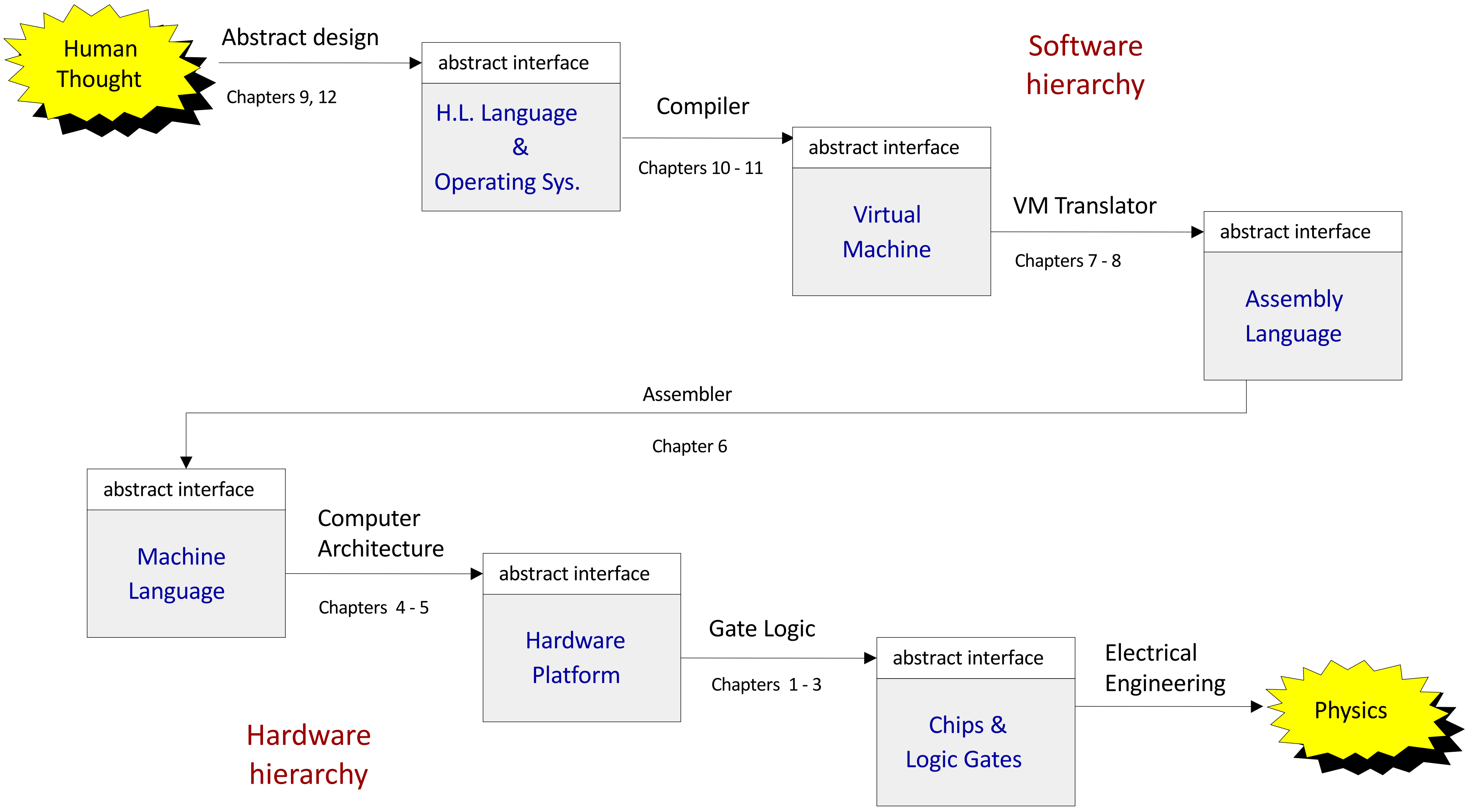



School of Computer Science

# COMP SCI 2000 / 7081 Computer Systems
# Lecture 2

# The whole system



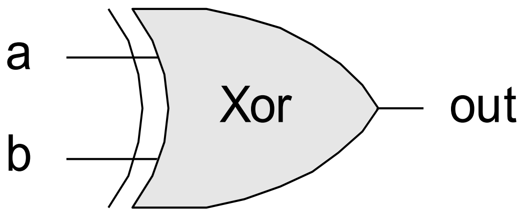

(Abstraction–implementation paradigm)

# Review: 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
  - Interfaces are a key abstraction mechanism

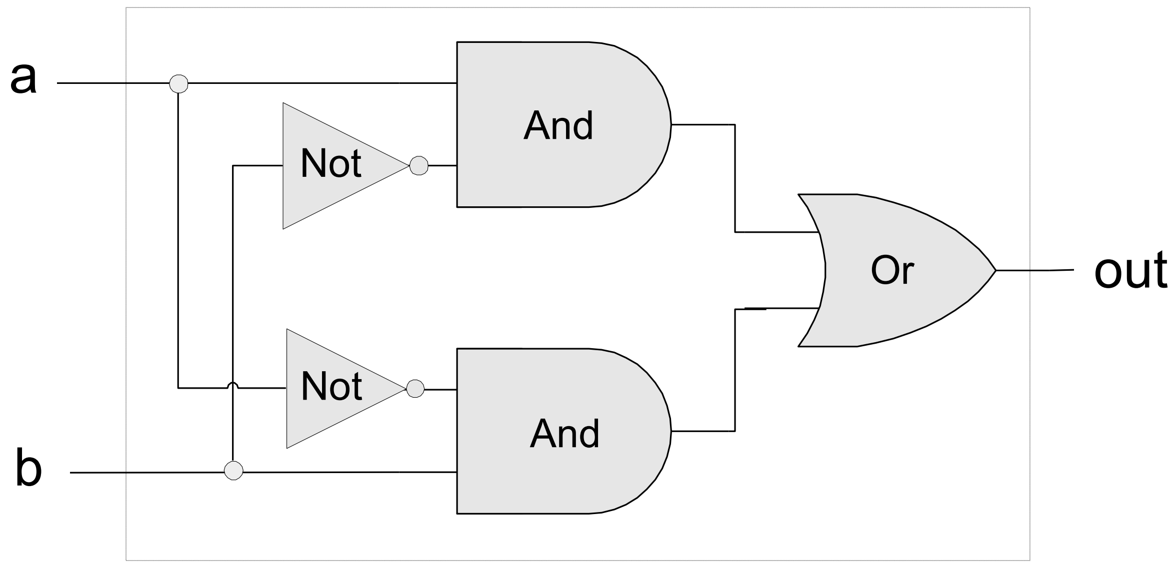
# Review: Example

## 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

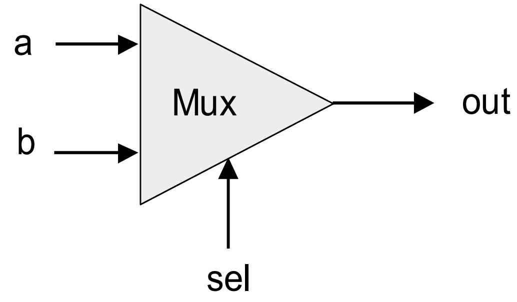
- We can construct a canonical representation of any boolean function
  - For each row that gives a 1 in its truth table
    - **and** together all terms after applying **not** to any 0 to make it a 1
    - if applied to any other row, the equation will evaluate to 0
  - Then
    - **or** together the equations for every row that gives a 1
- XOR
  - $\bar{x}.y + x.\bar{y}$

| 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
```

$$= \overline{a}.b.sel + a.\overline{b}.\overline{sel} + a.b.\overline{sel} + a.b.sel$$

$$= a.\overline{b}.\overline{sel} + a.b.\overline{sel} + a.b.sel + \overline{a}.b.sel$$

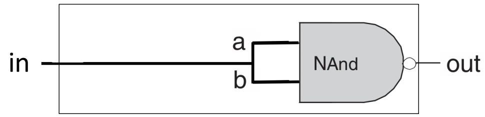
$$= a.\overline{sel} + b.sel$$

# How to construct **and**, **or** and **not**

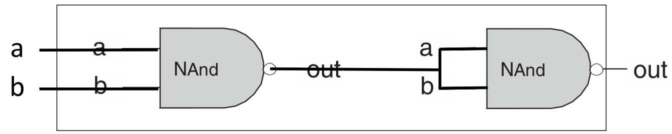
- From the truth table:
  - $\overline{x} = \overline{x.x}$
  - $x.y = \overline{\overline{x.y}}$
  - $x+y = \overline{\overline{x}.\overline{y}}$

| 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 |

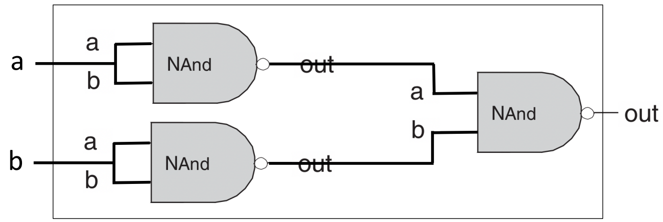
- Not(x)
  - Nand(x,x)




- And(x,y)
  - Nand(Nand(x,y),Nand(x,y))




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

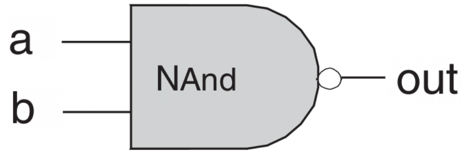



- We only need **nand** gates!

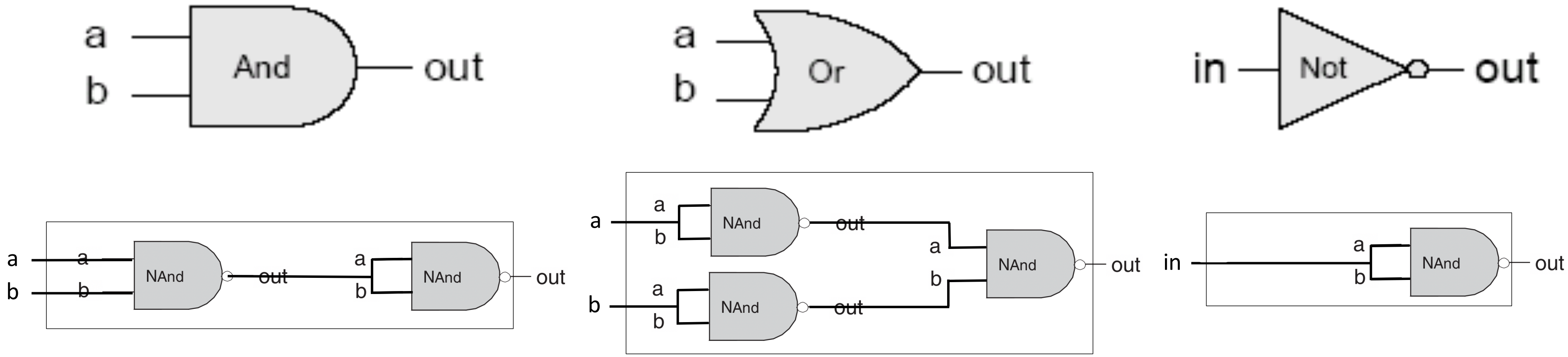
# Review: Gate logic

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

- Elementary gates:

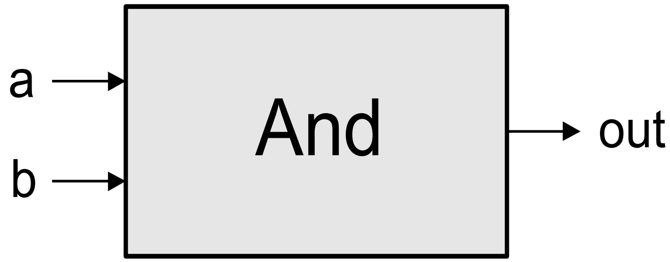



- Composite gates:




- <u>Important distinction:</u> 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

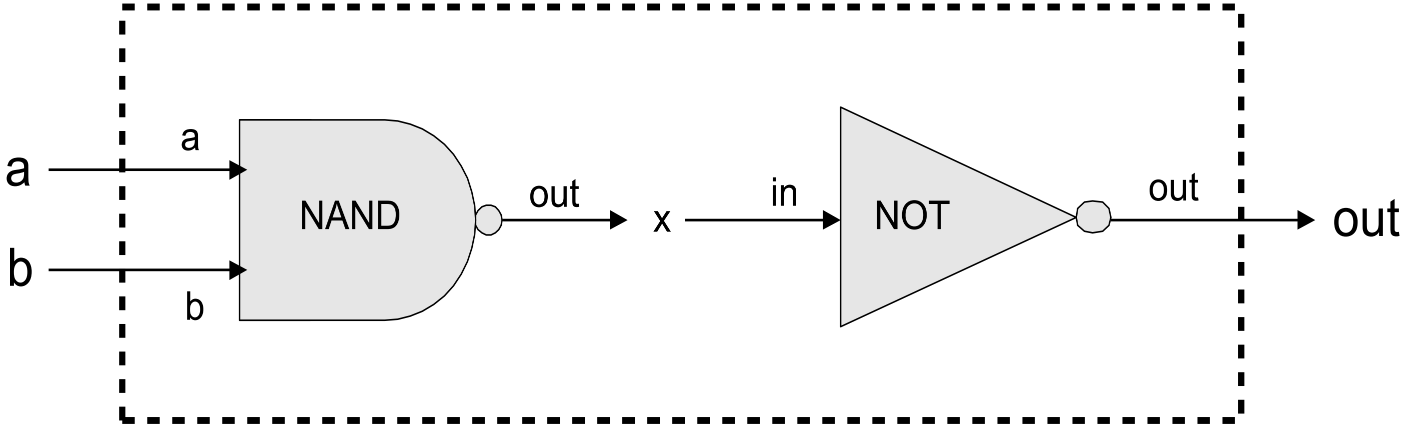
```
CHIP And
{   IN  a, b;
    OUT out;

    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))

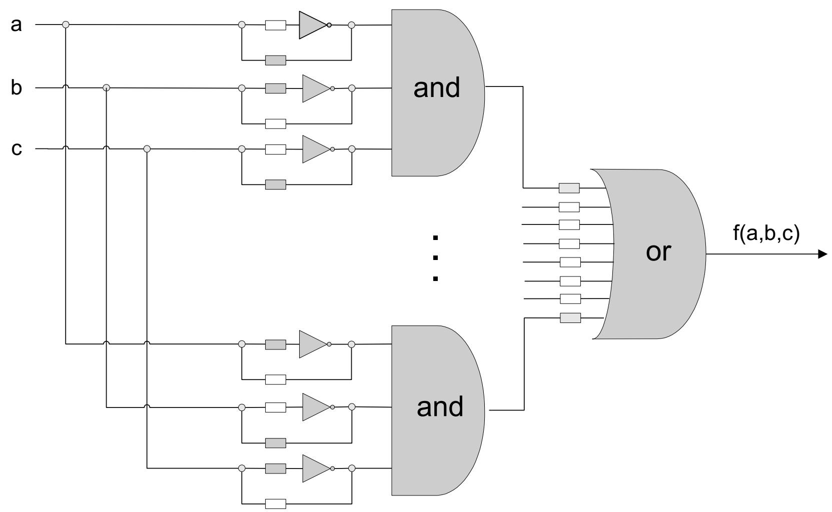



And.hdl `And(a = ?, b = ?, out = ?);`

```
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



# Number Representation

| Decimal | 4-bit 2's Complement | Decimal |
|---|---|---|
| -8 | 1000 | 8 |
| -7 | 1001 | 9 |
| -6 | 1010 | 10 |
| -5 | 1011 | 11 |
| -4 | 1100 | 12 |
| -3 | 1101 | 13 |
| -2 | 1110 | 14 |
| -1 | 1111 | 15 |
| 0 | 0000 | 0 |
| 1 | 0001 | 1 |
| 2 | 0010 | 2 |
| 3 | 0011 | 3 |
| 4 | 0100 | 4 |
| 5 | 0101 | 5 |
| 6 | 0110 | 6 |
| 7 | 0111 | 7 |

# Binary Addition

Assuming a 4-bit system:

| | | | | | |
|---|---|---|---|---|---|
| | 0 | 0 | 0 | 1 | |
| | 0 | 0 | 0 | 1 | + |
| | 0 | 1 | 0 | 1 | |
| 0 | 0 | 1 | 1 | 0 | |

No overflow 1 + 5 = 6

| | | | | | |
|---|---|---|---|---|---|
| | 1 | 1 | 1 | 1 | |
| | 1 | 0 | 1 | 1 | + |
| | 0 | 1 | 1 | 1 | |
| 1 | 0 | 0 | 1 | 0 | |

Overflow -5 + 7 = 2

- Algorithm: exactly the same as in decimal addition
- Overflow (MSB carry) may need to be dealt with – we usually ignore it.

# Binary Addition

- How do we know if a 2's complement number is negative?
  - The Most Significant Bit is 1
- There is only one representation of 0
- To negate a number, flip all the bits and add 1
- If you flip all the bits in a number x, you get -x – 1
- Sometimes the result of an add operation is wrong!
  - Using subtract to compare numbers needs to account for this effect

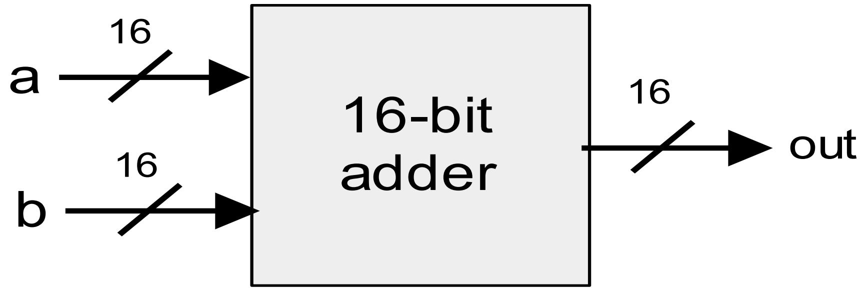
| | | | | | |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | | |
| | 1 | 0 | 1 | 1 | + |
| | 1 | 0 | 1 | 1 | |
| 1 | 0 | 1 | 1 | 0 | |

Bad overflow -5 + -5 = 6

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | | |
| | 0 | 1 | 0 | 1 | + |
| | 0 | 1 | 0 | 0 | |
| 0 | 1 | 0 | 0 | 1 | |

Bad overflow 5 + 4 = -7

# Building an Adder chip

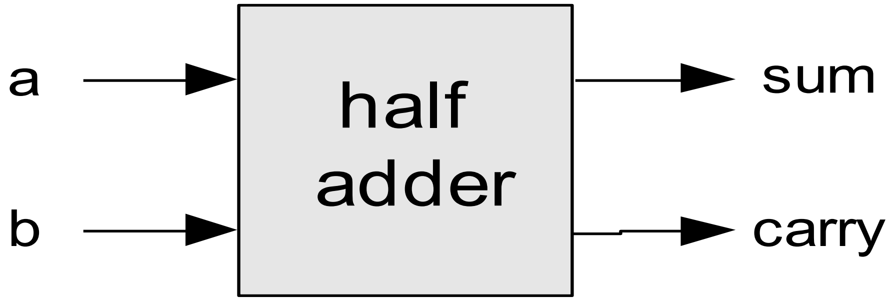



- Adder: a chip designed to add two integers

- Proposed implementation:
  - Half adder: designed to add 2 bits (we only need one)
  - Full adder: designed to add 3 bits (we need n-1 of these)
  - Adder: designed to add two *n*-bit numbers.

# Half adder (designed to add 2 bits)

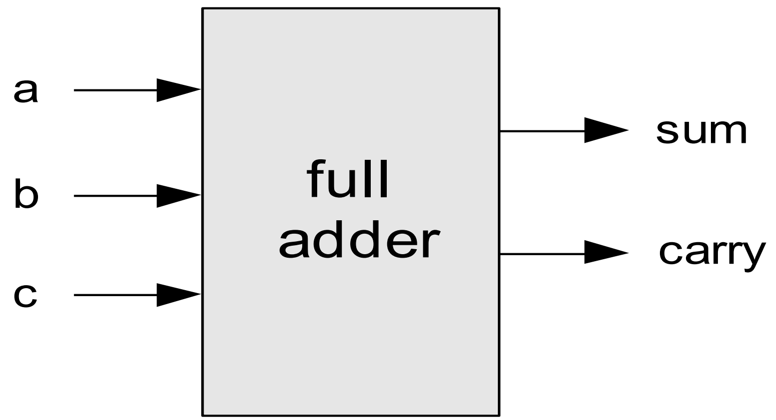
| a | b | sum | carry |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |




- A half adder can be built from an **xor** and an **and**
  - the sum column matches xor
  - the carry columns matches and

# Full adder (designed to add 3 bits)

| a | b | c | sum | carry |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |




<u>Implementation:</u> can be based on half-adder gates.

# Perspective

- <u>Combinational</u> logic
- The canonical representation would be too big to use
- Our adder design is very basic: no parallelism
- It pays to optimize adders (but we won't do that here)
- Where is the seat of more advanced maths operations?
  a typical hardware/software tradeoff.

# Summary

- You can construct many gates from NAND – this is just one example of how gates are built up.
- By understanding arithmetic, we can combine gates to add two numbers, then combine full-adders to add larger numbers.