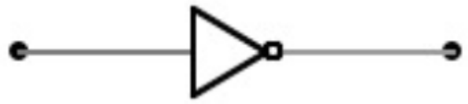
Revision – Part 1

Dr Tianxiang Cui

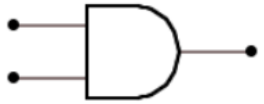
Boolean Logic

Elementary Logic Gates

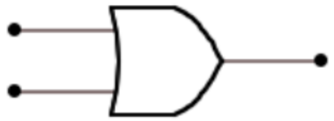
$A = \bar{A}$



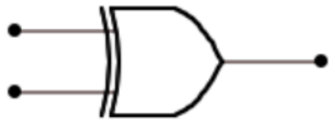
$A\ AND\ B = A \cdot B$



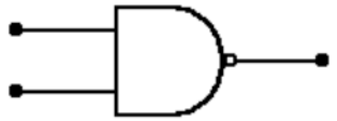
$A\ OR\ B = A + B$



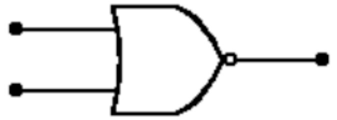
$A\ XOR\ B = A \oplus B$



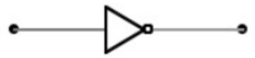
$A\ NAND\ B = \overline{A \cdot B}$



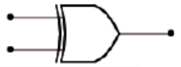
$A\ NOR\ B = \overline{A + B}$



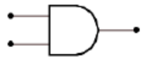
Collection of Elementary Logic Gates

$A = \bar{A}$ 

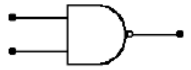
A	Ā
0	1

$A\ XOR\ B = A \oplus B$ 

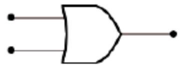
A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

$A\ AND\ B = A \cdot B$ 

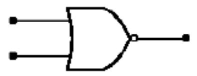
A	B	$A \bullet B$
0	0	0
0	1	0
1	0	0
1	1	1

$A\ NAND\ B = \overline{A \cdot B}$ 

A	B	$\overline{A \bullet B}$
0	0	1
0	1	1
1	0	1
1	1	0

$A\ OR\ B = A + B$ 

A	B	$A + B$
0	0	0
0	1	1
1	0	1
1	1	1

$A\ NOR\ B = \overline{A + B}$ 

A	B	$\overline{A + B}$
0	0	1
0	1	0
1	0	0
1	1	0

Boolean Logic

All chips constructed from elementary logic gates

- Every chip can be built from a combination of:
  - AND
  - OR
  - NOT
  - No integration, division, differentiation…
  - "Canonical Representation"
- AND, OR and NOT can be built from NAND

- Therefore every possible chip can be built from just the NAND gates!!!!

George Boole, 1815-1864

("*A Calculus of Logic*")

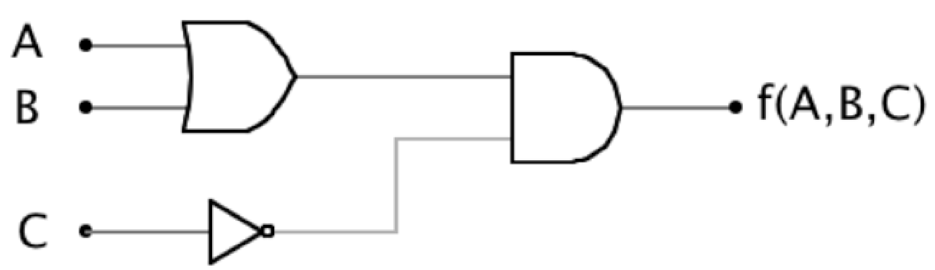
Boolean Function

- A Boolean function is a function that operates on binary inputs and return binary outputs
- Truth table is **every possible function evaluation** of the input variables
- [note 0 and 1 used to define false and true]
- Everything can be defined by a truth table

Composite Gates

$$f\left(A,B,C\right)=\left(A+B\right)\bullet\overline{C}$$

(A OR B) AND NOT C



A	B	C	*f*(A,B,C)
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

Precedence

- Precedence

  Parentheses evaluated first

  Then **Not**

  Then **And**

  Then **Or**

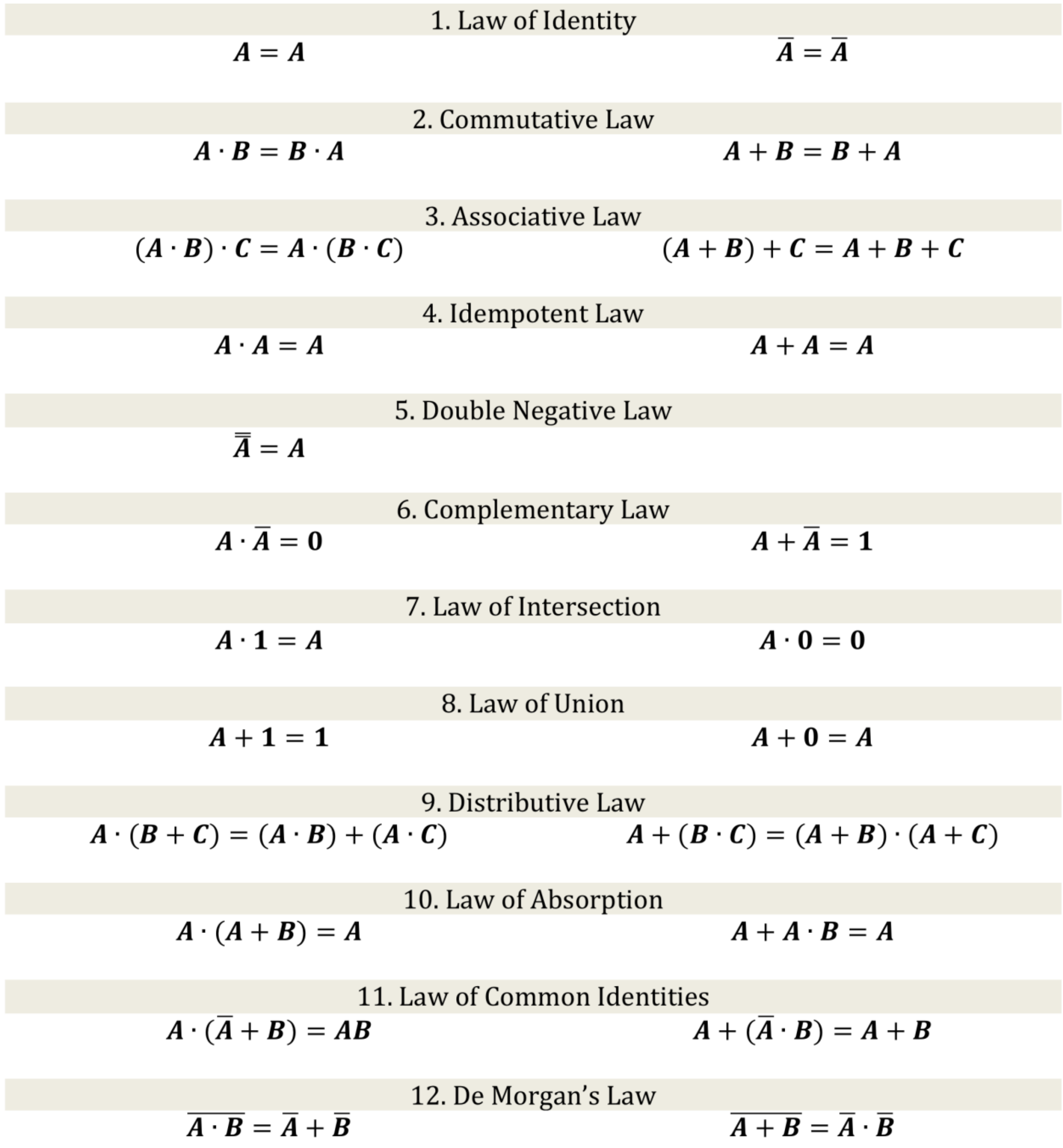
Not X Or Y And Z = (Not X) Or (Y And Z)

Not X And Y Or Z = ((Not X) And Y) Or Z

Brackets over-rule everything…use when in doubt

((Not (X)) And (Y)) Or (Z)

Laws of Boolean Algebra



1. Law of Identity

$\boldsymbol{A = A}$ $\qquad$ $\boldsymbol{\overline{A} = \overline{A}}$

2. Commutative Law

$\boldsymbol{A \cdot B = B \cdot A}$ $\qquad$ $\boldsymbol{A + B = B + A}$

3. Associative Law

$\boldsymbol{(A \cdot B) \cdot C = A \cdot (B \cdot C)}$ $\qquad$ $\boldsymbol{(A + B) + C = A + B + C}$

4. Idempotent Law

$\boldsymbol{A \cdot A = A}$ $\qquad$ $\boldsymbol{A + A = A}$

5. Double Negative Law

$\boldsymbol{\overline{\overline{A}} = A}$

6. Complementary Law

$\boldsymbol{A \cdot \overline{A} = 0}$ $\qquad$ $\boldsymbol{A + \overline{A} = 1}$

7. Law of Intersection

$\boldsymbol{A \cdot 1 = A}$ $\qquad$ $\boldsymbol{A \cdot 0 = 0}$

8. Law of Union

$\boldsymbol{A + 1 = 1}$ $\qquad$ $\boldsymbol{A + 0 = A}$

9. Distributive Law

$\boldsymbol{A \cdot (B + C) = (A \cdot B) + (A \cdot C)}$ $\qquad$ $\boldsymbol{A + (B \cdot C) = (A + B) \cdot (A + C)}$

10. Law of Absorption

$\boldsymbol{A \cdot (A + B) = A}$ $\qquad$ $\boldsymbol{A + A \cdot B = A}$

11. Law of Common Identities

$\boldsymbol{A \cdot (\overline{A} + B) = AB}$ $\qquad$ $\boldsymbol{A + (\overline{A} \cdot B) = A + B}$

12. De Morgan's Law

$\boldsymbol{\overline{A \cdot B} = \overline{A} + \overline{B}}$ $\qquad$ $\boldsymbol{\overline{A + B} = \overline{A} \cdot \overline{B}}$

Simplify Boolean Expression

Not(Not(*x*) And Not(*x* Or *y*)) =

Not(Not(*x*) And (Not(*x*) And Not(*y*))) =

Not((Not(*x*) And Not(*x*)) And Not(*y*)) =

Not(Not(*x*) And Not(*y*)) =

Not(Not(x Or y))= double negation

$\downarrow$

x Or *y*

Boolean Arithmetic

Binary to Decimal

- Each binary digit corresponds to a power of 2:

Place	7th	6th	5th	4th	3rd	2nd	1st	0th
Weight	2^7 $= 128$	2^6 $= 64$	2^5 $= 32$	2^4 $= 16$	2^3 $= 8$	2^2 $= 4$	2^1 $= 2$	2^0 $= 1$

- Where the digit is 1, we add the corresponding weight
- Example: convert 1100 1010_2 into decimal

$$\begin{aligned} 1100\,1010_2 &= 1 \times 128 + 1 \times 64 + 0 \times 32 + 0 \times 16 \\ &\quad + 1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1 \\ &= 128 + 64 + 8 + 2 = 202_{10} \end{aligned}$$

Decimal to Binary

- Repeatedly divide by 2, until we reach 0
- The **right**/**left**-most binary digit is the **first**/**last** remainder
- E.g. $101_{10} = 1100101_2$

101	Remainder
50	1
25	0
12	1
6	0
3	0
1	1
0	1

- Example: convert 163_{10} into binary
- 10100011_2

Decimal to Binary (look-up table)

- 87 = 64 (64 = 2^6 , the biggest 2^n that 87 is divisible by) + 23 (reminder)
- 87 = 64 + 16 (16 = 2^4 , the biggest 2^n that 23 is divisible by) + 7 (reminder)
- 87 = 64 + 16 + 4 (4 = 2^2 , the biggest 2^n that 7 is divisible by) + 3 (reminder)
- 87 = 64 + 16 + 4 + 2 (2 = 2^1 , the biggest 2^n that 3 is divisible by) + 1 (reminder)
- 87 = 64 + 16 + 4 + 2 + 1(1 = 2^0 , the biggest 2^n that 1 is divisible by) + 0 (reminder)
- Stop when reminder = 0

Representing Negative Numbers

- So far, unsigned numbers
  - How are negative numbers represented on a computer?
- What we use in decimal notation
  - +/− and 0, 1, 2, · · ·
- Such a representation is called **sign and magnitude**
- For binary numbers – define **leftmost** bit to be the **sign**
  - $0 \Rightarrow +$, $1 \Rightarrow -$
  - Rest of bits can be numerical value of number
  - Hence, only seven bits are left in a byte (apart from the sign bit), the magnitude can range from 0000000 (0) to 1111111 (127)
- Problems?

One’s Complement

- Alternatively, a system known as **one’s complement** can be used to represent negative numbers
- A negative binary number is the bitwise **NOT** applied to it — the "**complement**" of its positive counterpart
- E.g. the ones' complement form of 00101011 (43_{10}) becomes 11010100 (-43_{10})
- Still has two representations of 0: 00000000 (+0) and 11111111 (−0)
- The range of signed numbers using one’s complement is represented by $-(2^{N-1} - 1)$ to $(2^{N-1} - 1)$ and ± 0
  - A conventional eight-bit byte is -127_{10} to $+127_{10}$ with zero being either 00000000 (+0) or 11111111 (−0)

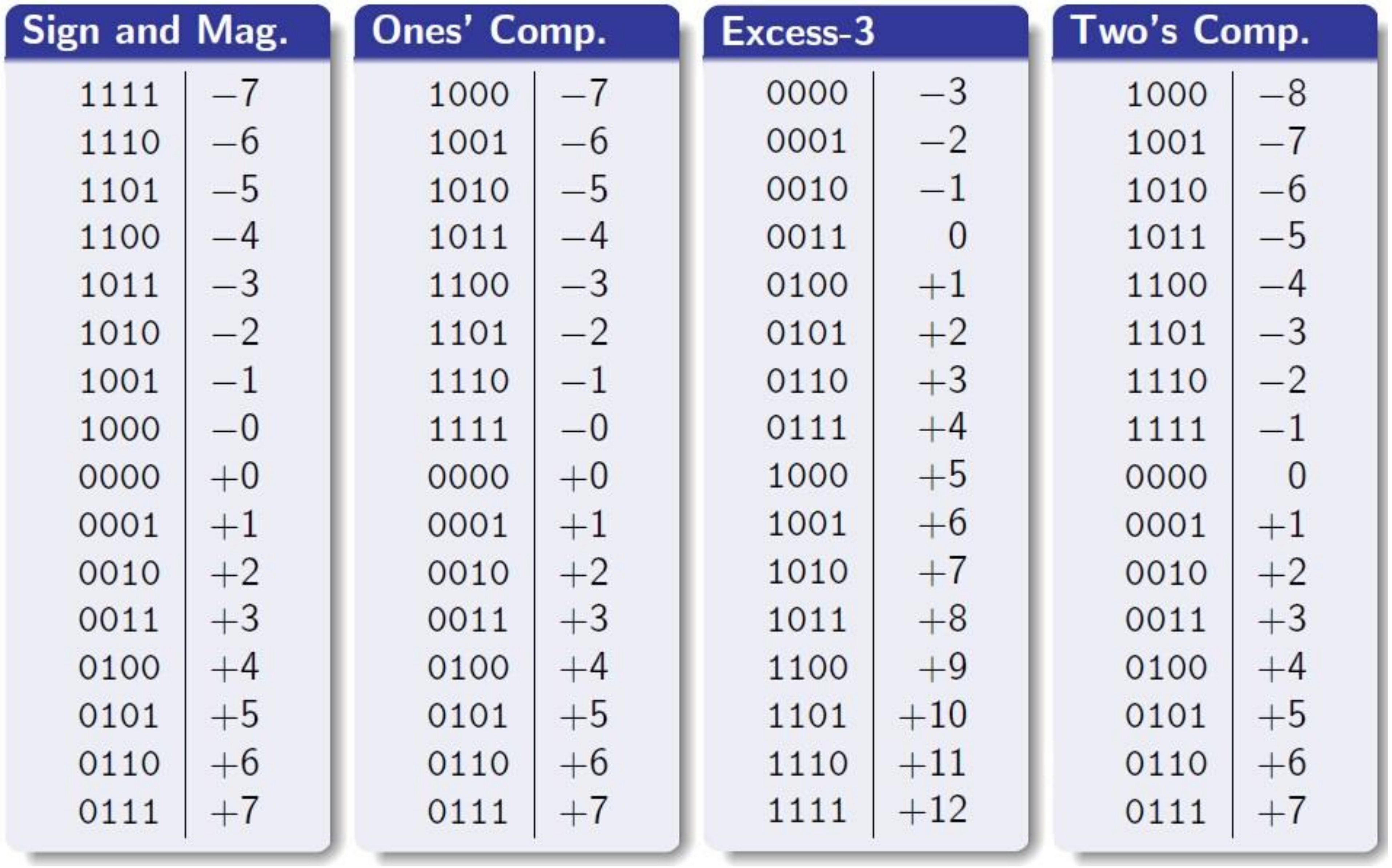
Excess-n

- **Excess-n**, also called offset binary or biased representation, uses a pre-specified number *n* as a biasing value
- A value is represented by the unsigned number which is *n* greater than the intended value
- Therefore 0 is represented by *n*, and −*n* is represented by the all-zeros bit pattern
- E.g. Excess-3
  - 0 is represented by 0011 (3)
  - +1 is represented by 0100 (4), +2 is represented by 0101(5)…
  - -1 is represented by 0010 (2), -2 is represented by 0001 (1)
  - -3 is represented by 0000 (0)

Two’s Complement

- The **two's complement** of an N-bit binary number is defined as the complement with respect to 2^N
  - It is the result of subtracting the number from 2^N
  - -x is represented as 2^N-x
- There’s a quicker way to calculate 2^N-x:
  - x + (1’s complement of x) = 2^N-1 (all 1 bits)
  - 2^N-x = (1’s complement of x) +1
  - Take the bitwise inverse (**NOT**) of x, then add 1 to result
- An N-bit two's-complement numeral system can represent every integer in the range $-(2^{N-1})$ to $+(2^{N-1} - 1)$
  - One’s complement: $-(2^{N-1} - 1)$ to $(2^{N-1} - 1)$
- The sum of a number and its two's complement will always equal 0 (the last digit is ignored)
  - The sum of a number and its one's complement will always equal -0 (all 1 bits)

Example of 4-Bit Signed Encodings



Sign and Mag.	
1111	-7
1110	-6
1101	-5
1100	-4
1011	-3
1010	-2
1001	-1
1000	-0
0000	$+0$
0001	$+1$
0010	$+2$
0011	$+3$
0100	$+4$
0101	$+5$
0110	$+6$
0111	$+7$

Ones' Comp.	
1000	-7
1001	-6
1010	-5
1011	-4
1100	-3
1101	-2
1110	-1
1111	-0
0000	$+0$
0001	$+1$
0010	$+2$
0011	$+3$
0100	$+4$
0101	$+5$
0110	$+6$
0111	$+7$

Excess-3	
0000	-3
0001	-2
0010	-1
0011	0
0100	$+1$
0101	$+2$
0110	$+3$
0111	$+4$
1000	$+5$
1001	$+6$
1010	$+7$
1011	$+8$
1100	$+9$
1101	$+10$
1110	$+11$
1111	$+12$

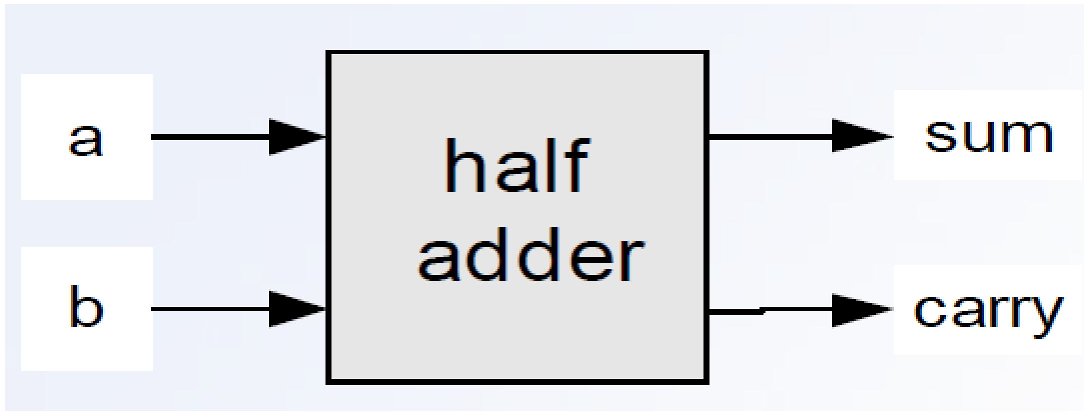
Two's Comp.	
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1
0000	0
0001	$+1$
0010	$+2$
0011	$+3$
0100	$+4$
0101	$+5$
0110	$+6$
0111	$+7$

Adder

- Build an Adder:
  - Half adder: adds two bits
  - Full adder: adds three bits
  - Adder: adds two integers

Half Adder

- Add **two** single binary digits and provide the **output** plus a **carry value**
- It has two inputs, called A(a) and B(b), and two outputs S (sum) and C (carry)



Half Adder

- Least significant bit in the addition is called sum (a+b)
- Most significant bit is called carry (carry of a+b)

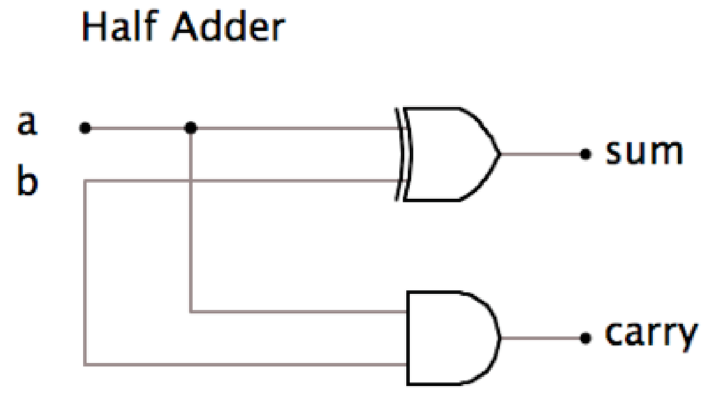
a	b	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

- **Never has a situation when sum and carry are both 1**

Half Adder

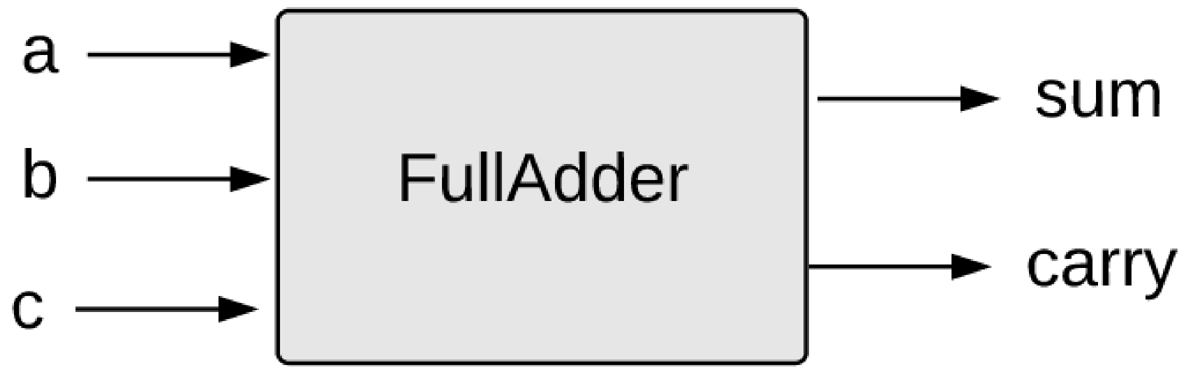
a	b	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

- The common representation uses a XOR and a AND gate



Full Adder

- Add **three** single binary digits and provide the **output** plus a **carry value**
- It has three inputs, called A, B and Carry(in), and two outputs S (sum) and Carry(out)



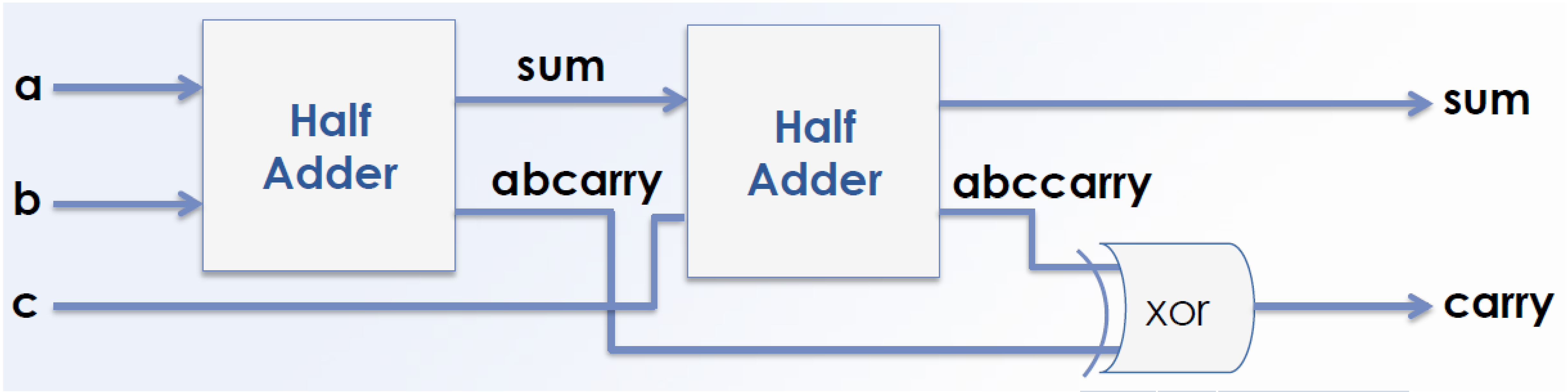
Full Adder

- Least significant bit in the addition is called sum (a+b+c_in)
- Most significant bit is called carry(out) (carry of a+b+c_in)

a	b	Carry(in)	Carry(out)	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

Full Adder: Implementation

- Use two half adders to build a full adder



A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Sequential Logic and ALU

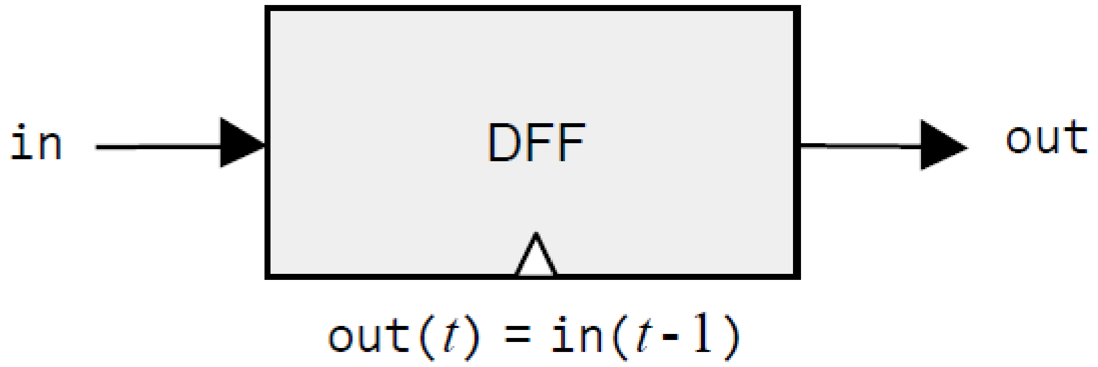
Sequential Logic Circuits

- Combinational chips compute functions that **depend solely on combinations of their input values**

- Sequential Logic Circuits
  - Output depends **not only on the present value** of its input signals but **on the sequence of past inputs**, the input history as well

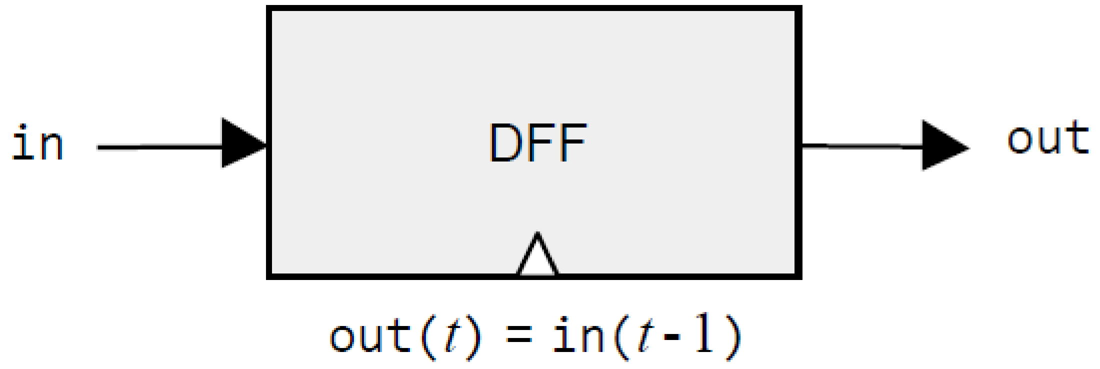
Flip Flops

- The flip flop is the most elementary sequential element in the computer
- Data Flip Flop (DFF): the simplest state keeping gate (built-in)



- Contains a **single** bit **input** and a **single** bit **output**

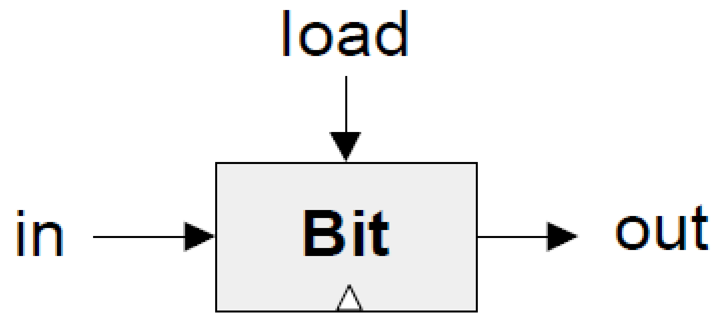
Flip Flops



- The gate outputs its previous input: out(t)= in(t-1)
- Implementation: a gate that can flip between two stable states:
  - Remembering 0/Remembering 1
  - Also can be made from looping NAND gates

Register

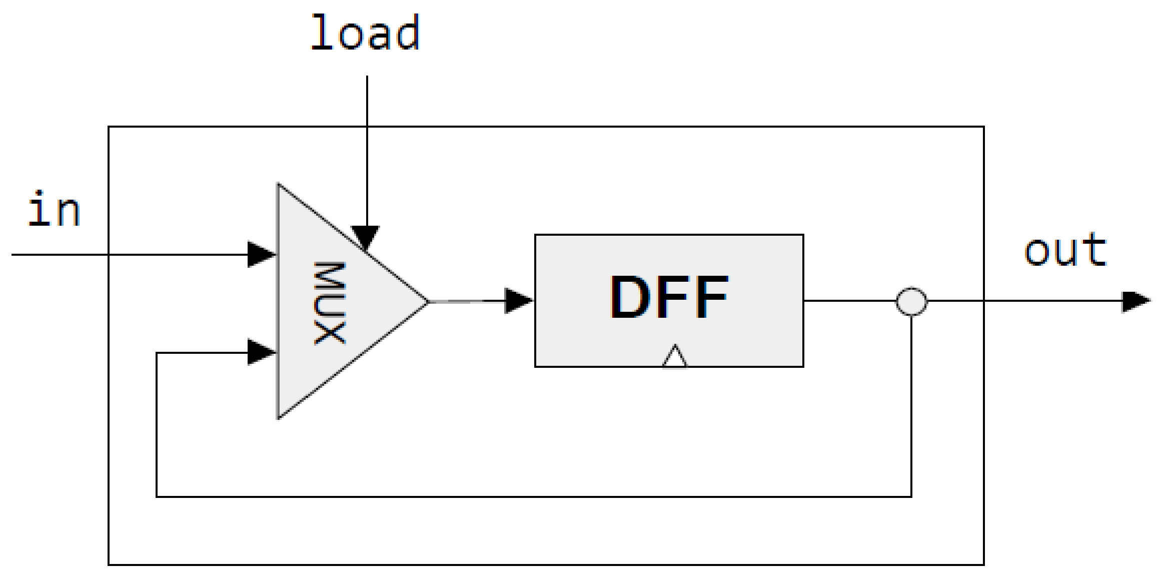
- A register is a storage device that can "**store**" or "**remember**" a value over time
- Typically is composed of flip flops
- 1-bit register:
  - Store (maintain) a bit
  - Until it is instructed to load(store) another bit



if `load`(t) then `out`$(t+1)$ = `in`(t)
else `out`$(t+1)$ = `out`(t)

1-bit Register: Implementation

- The select bit of the Mux can become the load bit!

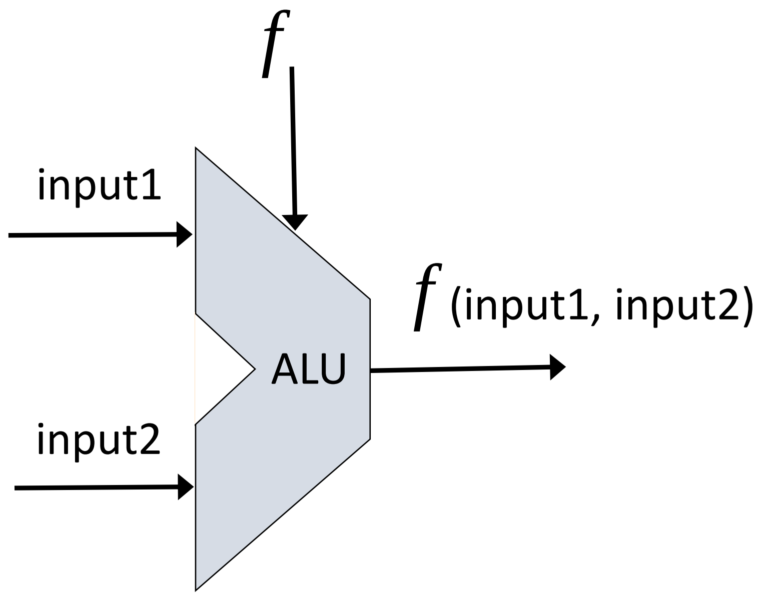


Arithmetic Logical Unit

- **A combinational circuit** that performs arithmetic and bitwise operations on integers represented as binary numbers.
- Input the **data** and some **code for the operation**
- Output will be some **data** and any **additional information**
- ALUs perform simple functions, because of this they can be executed at high speeds (i.e., very short propagation delays)

The Arithmetic Logical Unit

The ALU computes a function on the two inputs, and outputs the result



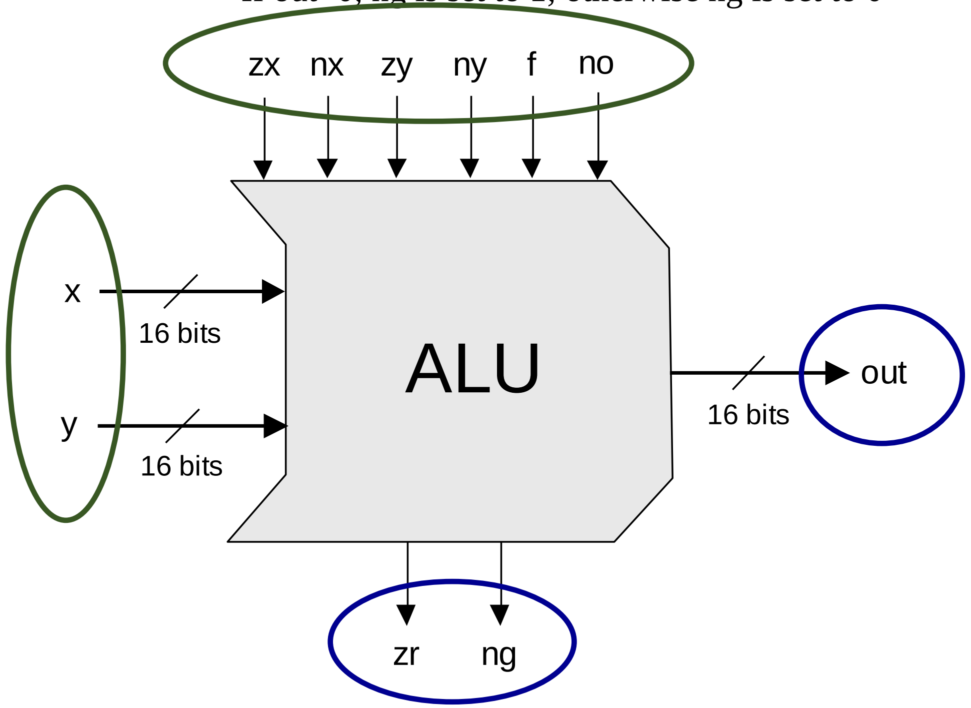
f: one out of a family of pre-defined arithmetic and logical functions

- Arithmetic functions: integer addition, multiplication, division, ...
- logical functions: And, Or, Xor, …

Which functions should the ALU perform?
A hardware / software tradeoff.

The Hack ALU

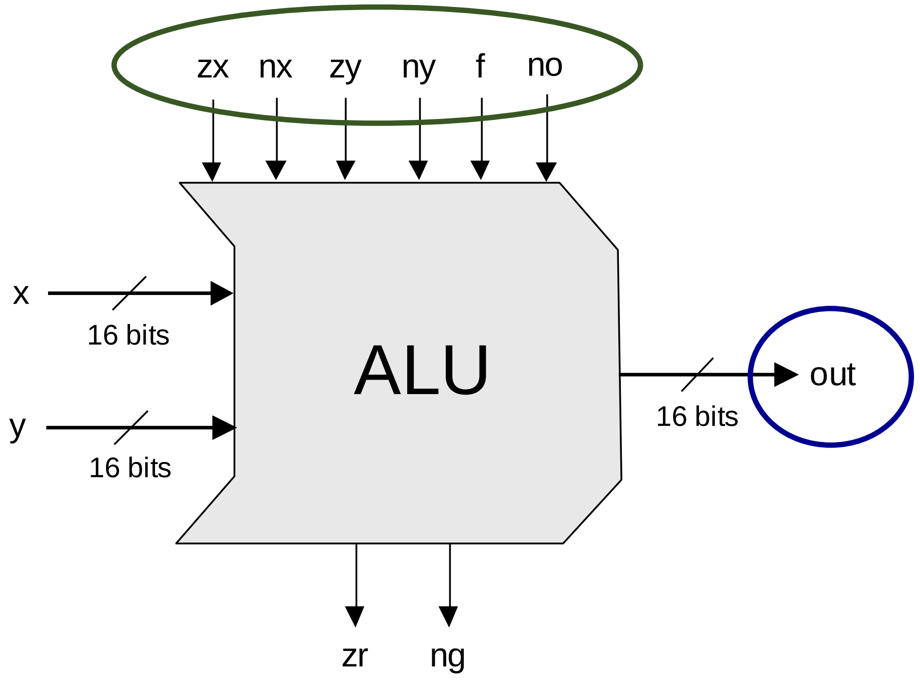
- Operates on two 16-bit, two's complement values
- Outputs a 16-bit, two's complement value
- Which function to compute is set by six 1-bit inputs
- Computes one out of a family of 18 functions
- Also outputs two 1-bit values
  - if the ALU output is 0, zr is set to 1; otherwise zr is set to 0
  - If out<0, ng is set to 1; otherwise ng is set to 0



out
0
1
-1
x
y
!x
!y
-x
-y
x+1
y+1
x-1
y-1
x+y
x-y
y-x
x&y
x|y

The Hack ALU

To cause the ALU to compute a function, set the control bits to one of the binary combinations listed in the table.



control bits

zx	nx	zy	ny	f	no	out
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

Memory

Fetch-Decode-Execute Cycle

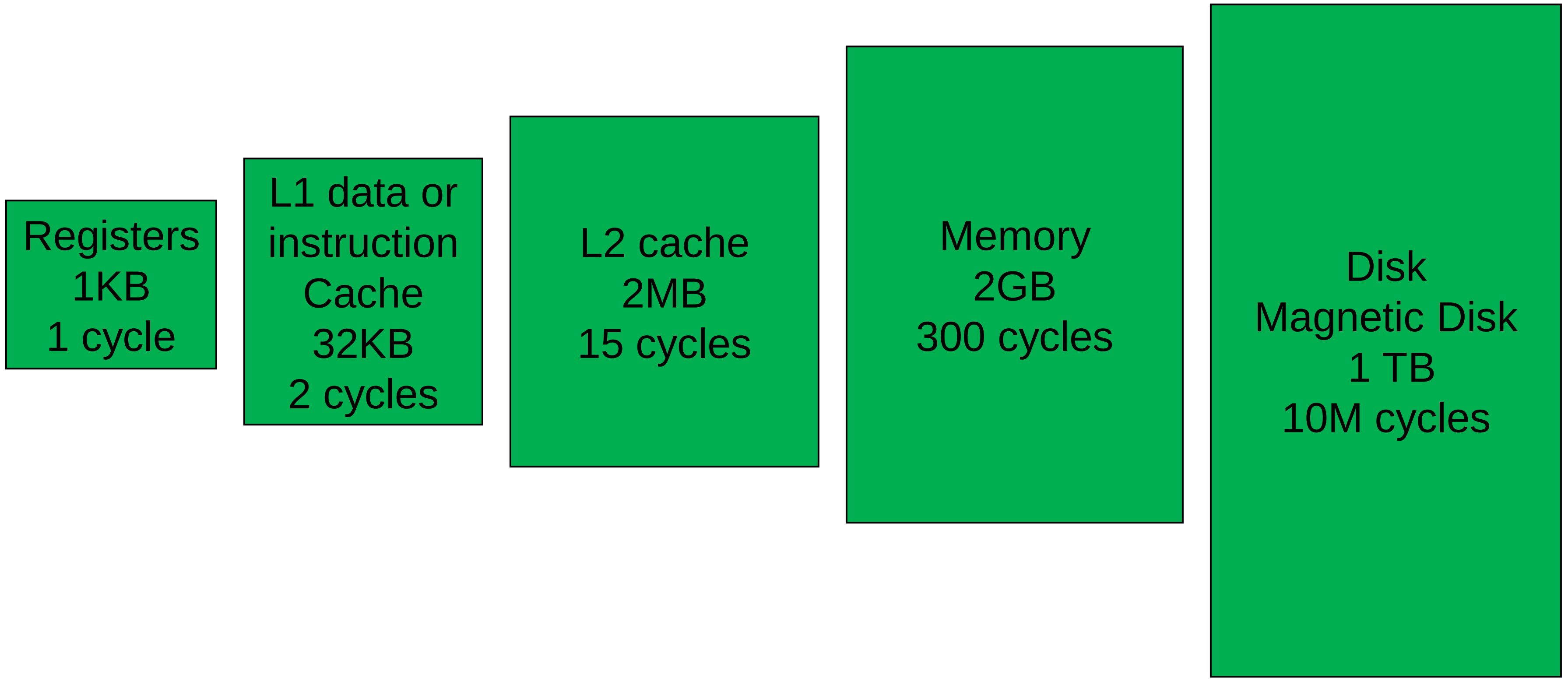
- At some level, every programmable processor implements a **fetch-execute cycle**
- Automatically implemented by processor hardware, allows processor to move through program steps
- Fetch — The opcode for the instruction is fetched from memory
- Decode — Opcode decoded to work what parts of the CPU are needed
- Execute — CPU processes the instruction
- And repeat for the next instruction

Fetch-Execute Algorithm

```
Repeat {
  Fetch (PC):
    • Fetch the instruction word (at PC)
    • Instruction decoded
    • Calculate next instruction address
  Execute (ALU, Registers and Control):
    • Read operands
    • Executes the operations
    • Write/store results
}
```

Memory Hierarchy

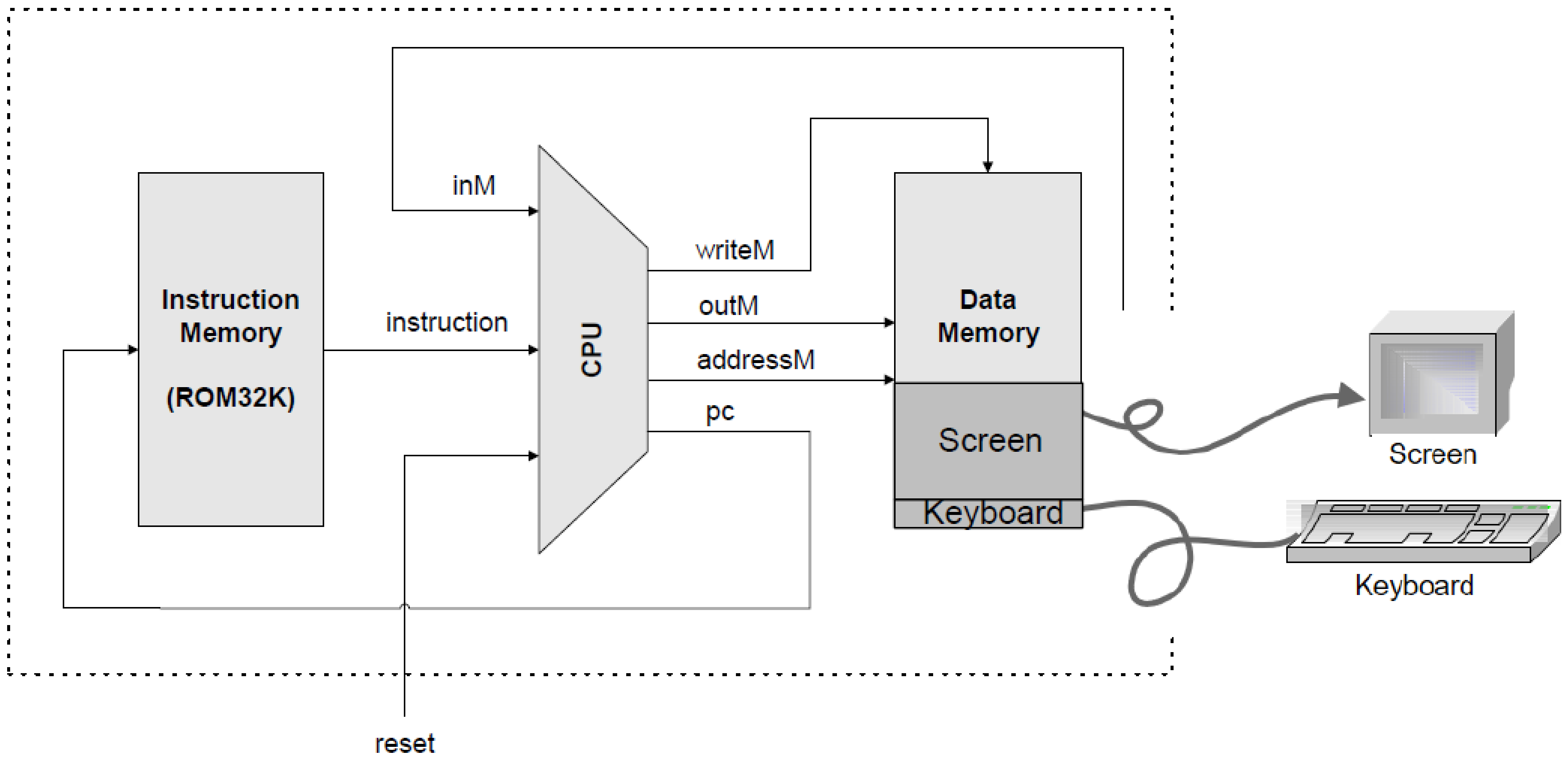
- As it goes further, capacity and latency increase



The Hack Computer: Main Parts

- Instruction memory (ROM)
- Memory (RAM)
    - Data memory
    - Screen (memory map)
    - Keyboard (memory map)
- CPU
- Computer (the logic that holds everything together)

The Hack Computer (Put Together)


inM
writeM
outM
addressM
pc
Instruction Memory
(ROM32K)
instruction
CPU
Data Memory
Screen
Keyboard
Screen
Keyboard
reset