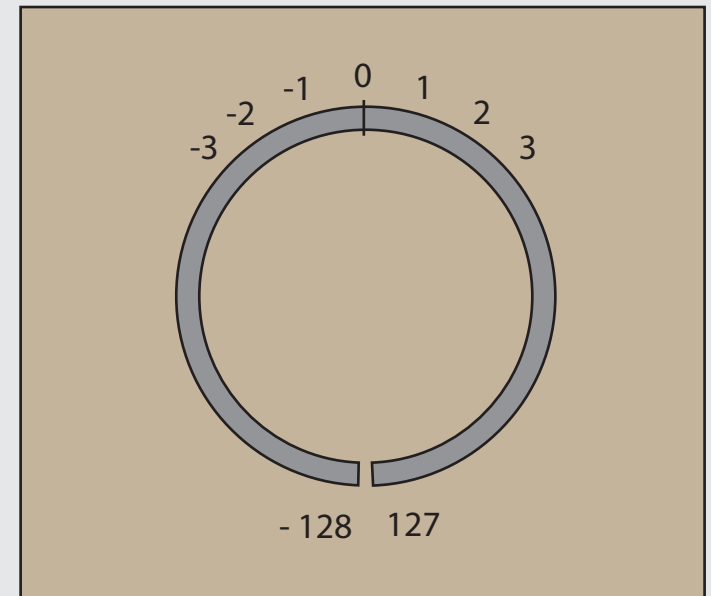


# Computer Architecture

Paul Mellies

Lecture 3 : Number Systems



# Decimal numeration

0	○
1	一
2	二
3	三
4	四
5	五
6	六
7	七
8	八
9	九

**Key idea :** ten numbers are sufficient to write down all the other natural numbers

**Method :** decompose any natural number  $n$  as a sum of powers of 10's

$$\begin{aligned}4717 &= 4000 + 700 + 10 + 7 \\ &= 4 \times 10^3 + 7 \times 10^2 + 1 \times 10^1 + 7 \times 10^0\end{aligned}$$

$$\begin{aligned}2020 &= 2000 + 0 + 20 + 0 \\ &= 2 \times 10^3 + 0 \times 10^2 + 2 \times 10^1 + 0 \times 10^0\end{aligned}$$

# How decimal numbers work

1's column  
10's column  
100's column  
1000's column

$$2020_{\text{ten}} = 2 \times 10^3 + 0 \times 10^2 + 2 \times 10^1 + 0 \times 10^0$$

two thousand's      zero hundreds      one tens      nine ones

# Roman numbers

I	1
II	2
III	3
IV	4
V	5
VI	6
VII	7
VIII	8
IX	9
X	10

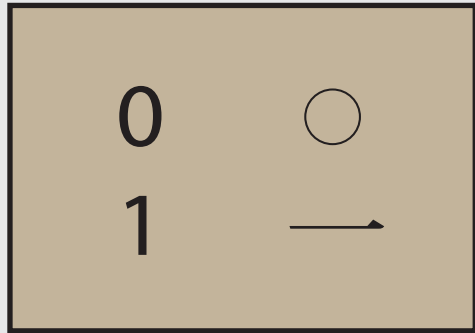
XI	11
XII	12
XIII	13
XIV	14
XV	15
XVI	16
XVII	17
XVIII	18
XIX	19
XX	20

X	10
XX	20
XXX	30
XL	40
L	50
LX	60
LXX	70
LXXX	80
XC	90
C	100

C	100
CC	200
CCC	300
CD	400
D	500
DC	600
DCC	700
DCCC	800
CM	900
M	1000

Difficult to add and to multiply and to divide with them...  
Also, difficult to write big numbers... or very small ones !

# Binary numeration



**Key idea :** the two numbers 0 and 1 are sufficient to encode all natural numbers

**Method :** decompose any natural number  $n$  as a sum of powers of 2's

$$\begin{aligned} 4717 &= 4096 && + 512 && + 64 & + 32 && + 8 & + 4 && + 1 \\ &= 1x2^{12} + 0x2^{11} + 0x2^{10} + 1x2^9 + 0x2^8 + 0x2^7 + 1x2^6 + 1x2^5 + 0x2^4 + 1x2^3 + 1x2^2 + 0x2^1 + 1x2^0 \end{aligned}$$

1	0	0	1	0	0	1	1	0	1	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---

$$\begin{aligned} 2020 &= 1024 + 512 + 256 + 128 + 64 + 32 && + 4 \\ &= 1x2^{10} + 1x2^9 + 1x2^8 + 1x2^7 + 1x2^6 + 1x2^5 + 0x2^4 + 0x2^3 + 1x2^2 + 0x2^1 + 0x2^0 \end{aligned}$$

1	1	1	1	1	1	0	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---

# How binary numbers work

1's column  
2's column  
4's column  
8's column  
16's column

$$10110_{\text{two}} = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 22_{\text{ten}}$$

one sixteen      no eight      one four      one two      no one

# Binary numbers and their decimal equivalent

1-Bit Binary Numbers	2-Bit Binary Numbers	3-Bit Binary Numbers	4-Bit Binary Numbers	Decimal Equivalents
0	00	000	0000	0
1	01	001	0001	1
	10	010	0010	2
	11	011	0011	3
		100	0100	4
		101	0101	5
		110	0110	6
		111	0111	7
			1000	8
			1001	9
			1010	10
			1011	11
			1100	12
			1101	13
			1110	14
			1111	15

# Exercise

Compute the decimal value of the following binary numbers :

10

11

0111

1110

1101

0110 1000

1001 0000 0000

1001 0110 1000

1 0000 0000 0000

1111 1111 1111

1111 1111 1110

1111 1101 1111



# Powers of 2

$2^0$	=	1
$2^1$	=	2
$2^2$	=	4
$2^3$	=	8
$2^4$	=	16
$2^5$	=	32
$2^6$	=	64
$2^7$	=	128
$2^8$	=	256
$2^9$	=	512
$2^{10}$	=	1 024

$2^{10}$	=	1 024
$2^{11}$	=	2 048
$2^{12}$	=	4 096
$2^{13}$	=	8 192
$2^{14}$	=	16 384
$2^{15}$	=	32 768
$2^{16}$	=	65 536
$2^{17}$	=	131 072
$2^{18}$	=	262 144
$2^{19}$	=	524 288
$2^{20}$	=	1 048 576

# Hexadecimal numeration

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

In practice, it is often convenient to group 0's and 1's into groups of four bits.

Each such group (also called nibble) defines a number between 0 and 15.

This leads to the hexadecimal numeration.

Note that every pair of hexadecimals defines a group of eight bits, called a byte:

$$\begin{aligned} 1A_{\text{hex}} &= 0001\ 1010_{\text{two}} = 26_{\text{ten}} \\ D5_{\text{hex}} &= 1101\ 0101_{\text{two}} = 213_{\text{ten}} \\ 47_{\text{hex}} &= 0100\ 0111_{\text{two}} = 71_{\text{ten}} \end{aligned}$$

This notation is very useful to denote the memory addresses of the computer.

# How hexadecimal numbers work

1's column  
16's column  
256's column

$$2ED_{\text{hex}} = 2 \times 16^2 + E \times 16^1 + D \times 16^0 = 749_{\text{ten}}$$

two  
two hundred  
fifty six's

fourteen  
sixteens

thirteen  
ones

# Hexadecimal numeration

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Recall that the value of a register in MIPS or a memory address in MIPS is a « word » typically defined as a sequence of 32 bits.

A word may be alternatively defined as a sequence of four bytes.

A byte is thus represented by a pair of hexadecimals :

$$0110\ 1010_{\text{two}} = 6A_{\text{hex}}$$

Similarly, a word is represented by a sequence of eight hexadecimals :

$$11111111\ 10000000\ 01101010\ 01110010_{\text{two}} \\ = \\ FF\ 80\ 6A\ 72_{\text{hex}}$$

The value of a register or a memory address in MIPS is thus represented by a sequence of eight hexadecimals.

# Exercise

Translate the following binary numbers into hexadecimal numbers :

10

11

0111

1110

1101

0110 1000

1001 0000 0000

1001 0110 1000

1 0000 0000 0000

1111 1111 1111

1111 1111 1110

1111 1101 1111

# Addition

# Addition between decimal numbers ( well-known )

traditional algorithm :  
addition with carry

$$\begin{array}{r} \phantom{+} 35 \\ + 57 \\ \hline 92 \end{array}$$

addition with carry

$$\begin{array}{r} \phantom{+} 4835 \\ + 9457 \\ \hline 14292 \end{array}$$

# Addition between binary numbers

addition with carry

$$\begin{array}{r} \phantom{+} \phantom{1} \phantom{1} \phantom{0} \\ \phantom{+} \phantom{1} \phantom{1} \phantom{0} \\ \phantom{+} \phantom{1} \phantom{1} \phantom{0} \\ + \phantom{1} \phantom{1} \phantom{0} \\ \hline 1 \phantom{1} \phantom{0} \phantom{1} \end{array}$$

The diagram shows the addition of two 3-bit binary numbers: 111 and 110. The result is 1101. Two carry bits are shown above the first two columns: a '1' above the first column and a '0' above the second column. Arched arrows point from the top of the first column to the top of the second column, and from the top of the second column to the top of the third column, indicating the carry propagation.

addition with carry

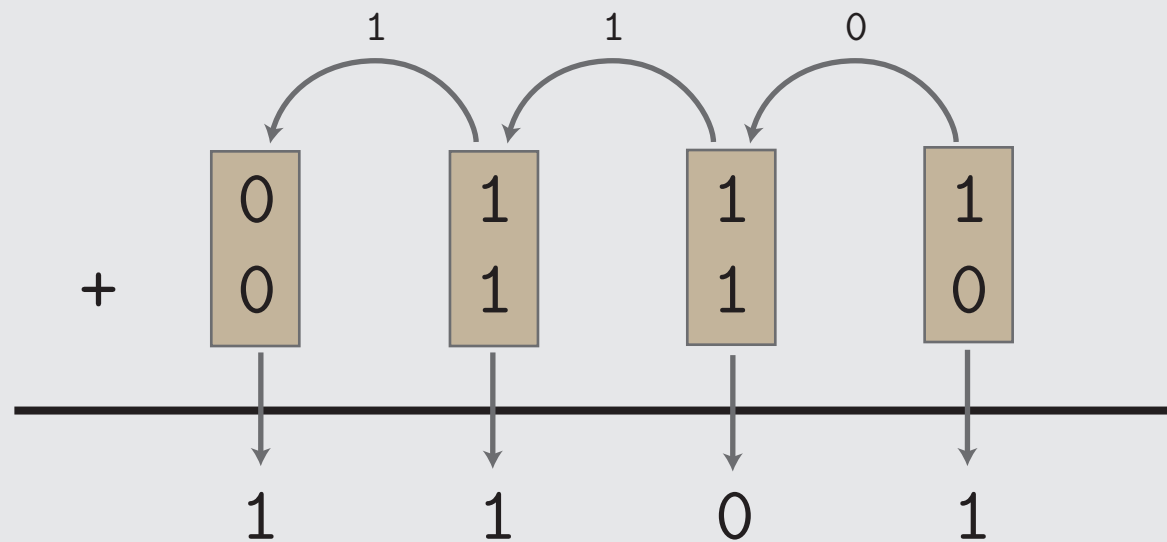
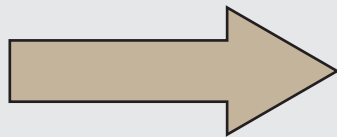
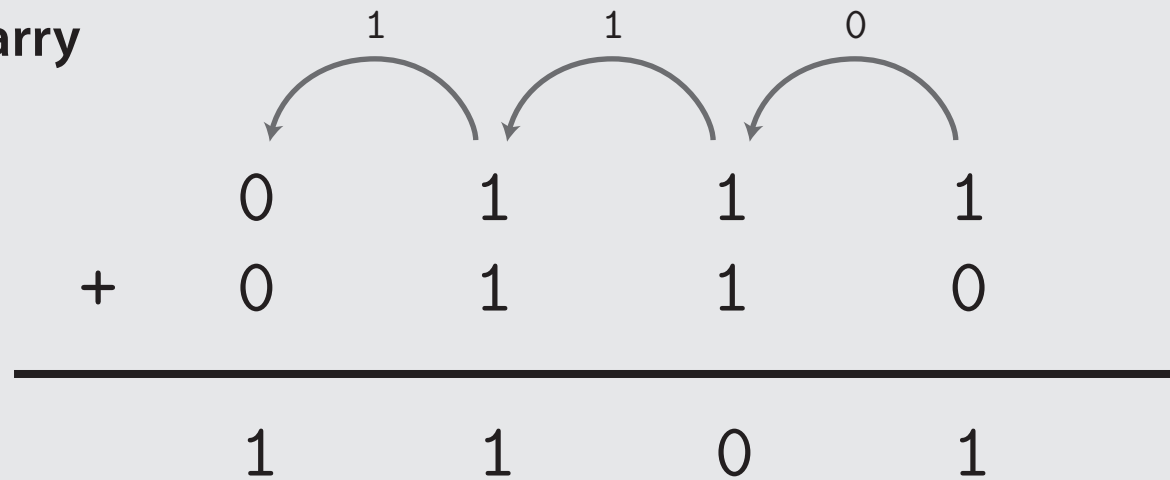
$$\begin{array}{r} \phantom{+} \phantom{0} \phantom{0} \phantom{0} \phantom{0} \phantom{0} \phantom{1} \phantom{1} \phantom{0} \\ \phantom{+} \phantom{0} \phantom{1} \phantom{1} \phantom{0} \phantom{0} \phantom{0} \phantom{1} \phantom{1} \phantom{1} \\ \phantom{+} \phantom{0} \phantom{0} \phantom{0} \phantom{0} \phantom{0} \phantom{0} \phantom{1} \phantom{1} \phantom{0} \\ + \phantom{0} \phantom{0} \phantom{0} \phantom{0} \phantom{0} \phantom{0} \phantom{0} \phantom{1} \phantom{1} \phantom{0} \\ \hline 0 \phantom{1} \phantom{1} \phantom{0} \phantom{0} \phantom{0} \phantom{1} \phantom{1} \phantom{0} \phantom{1} \end{array}$$

The diagram shows the addition of two 9-bit binary numbers: 011000111 and 000000110. The result is 011001101. Eight carry bits are shown above the columns: '0' above the first five columns, '1' above the sixth and seventh columns, and '0' above the eighth column. Arched arrows point from the top of each column to the top of the next column to the right, indicating the carry propagation.



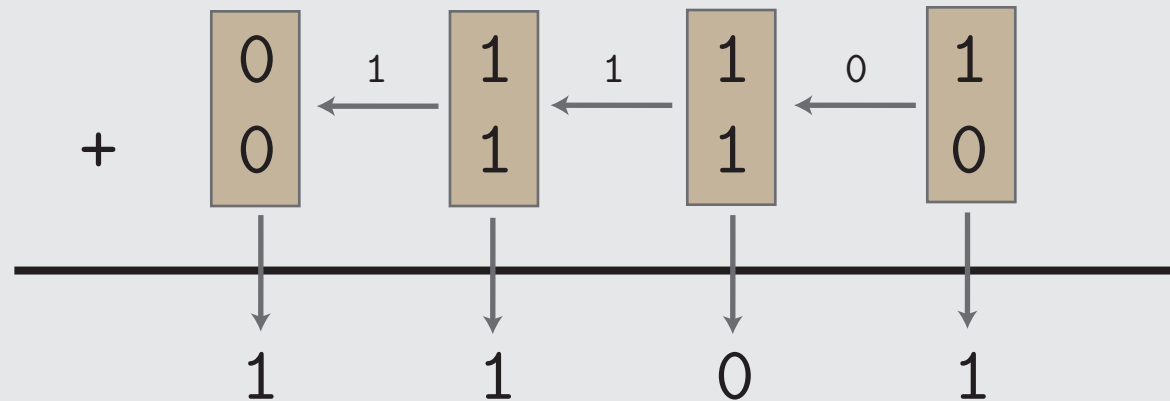
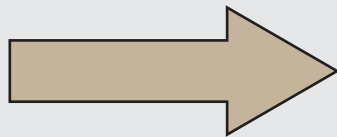
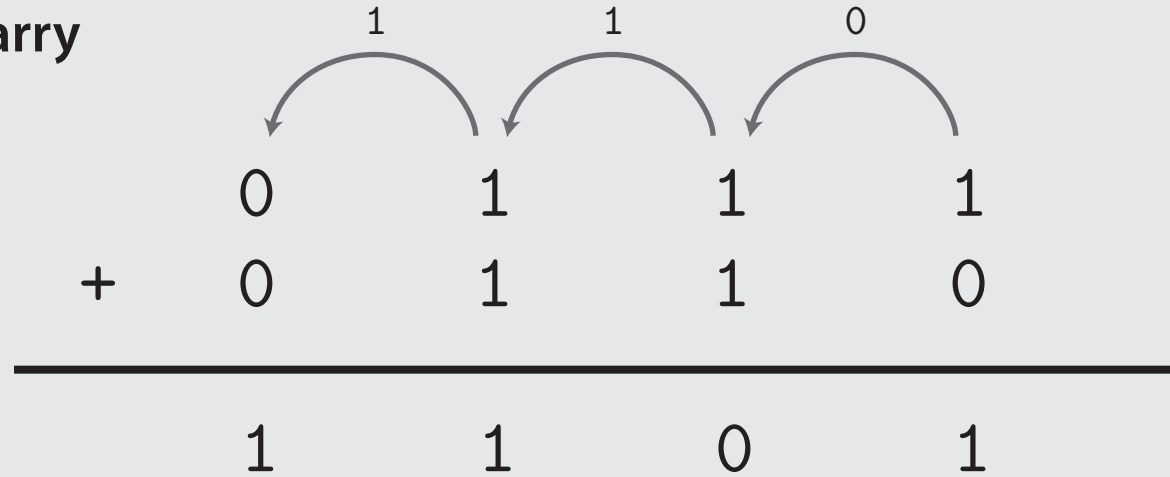
# Question : Can we implement this algorithm using logical gates ?

addition with carry

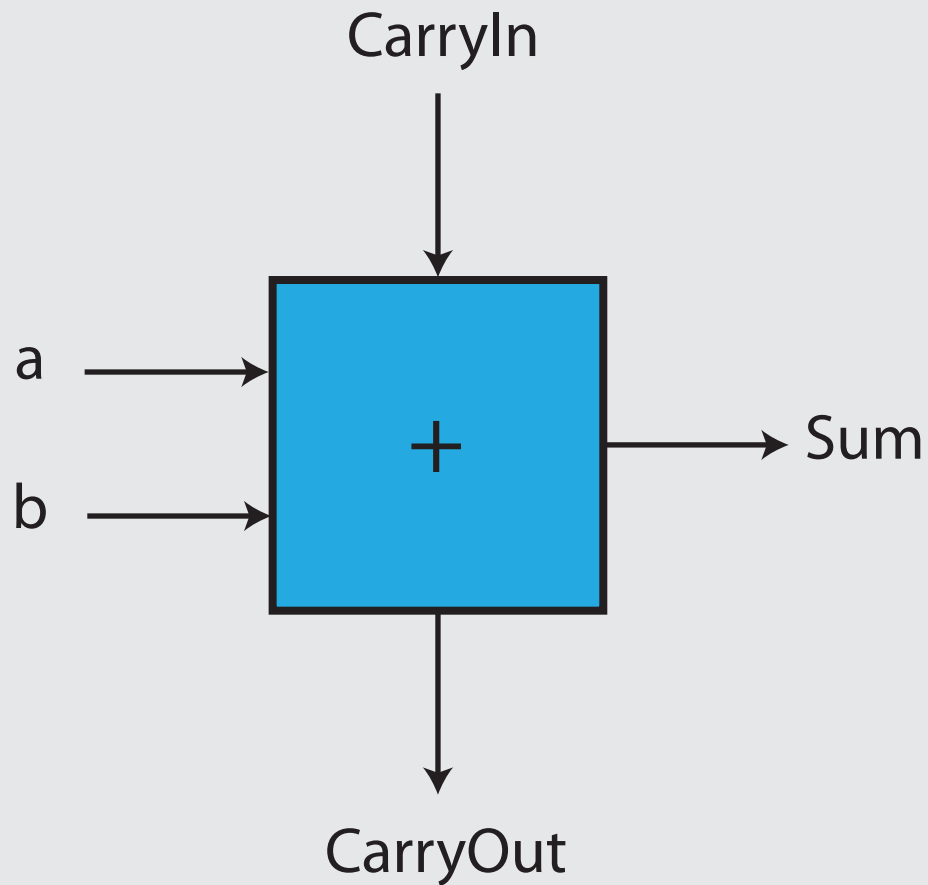


# Question : Can we implement this algorithm using logical gates ?

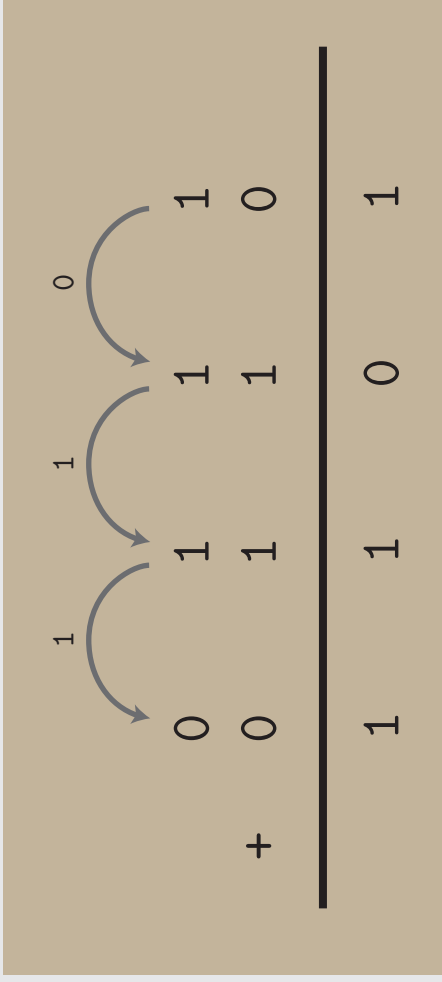
addition with carry



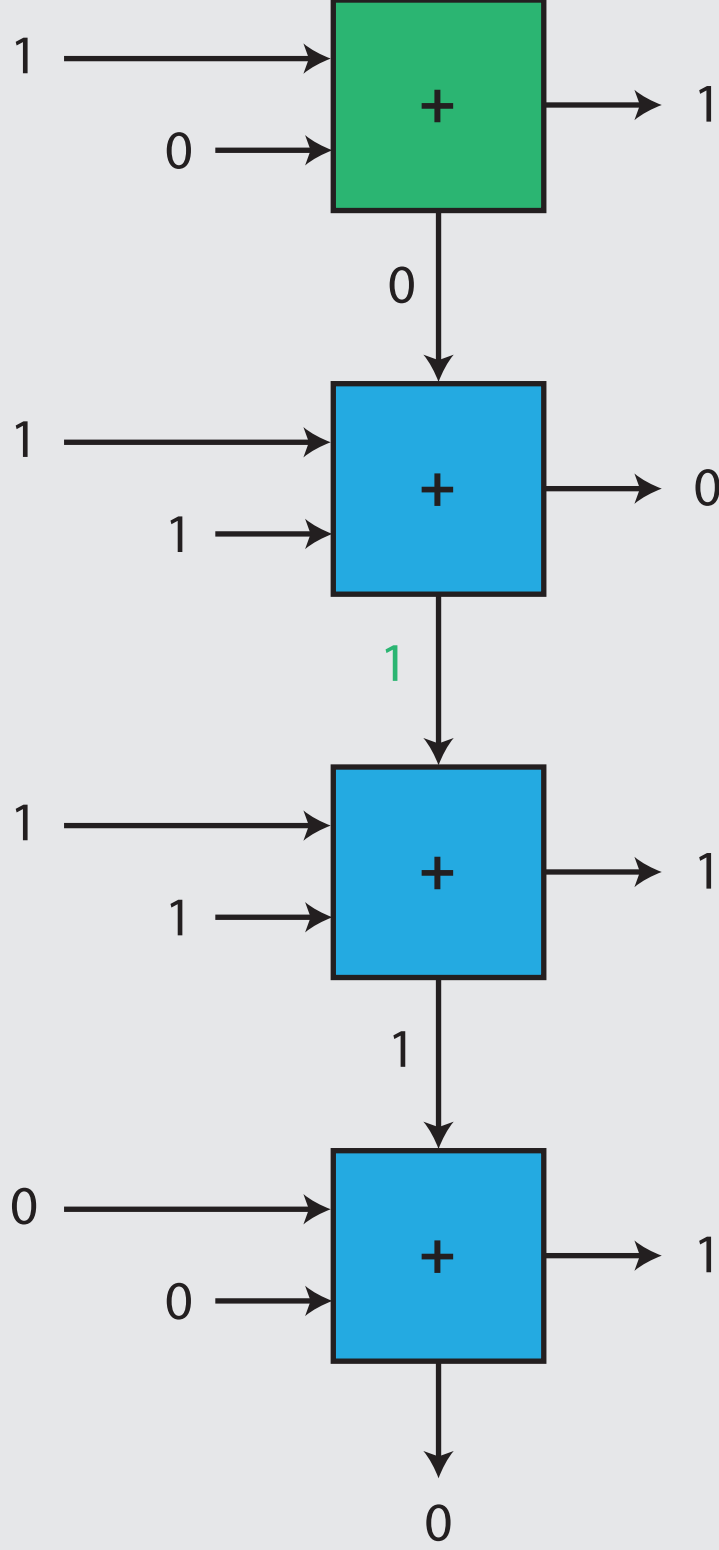
**Question : Can we implement this algorithm using logical gates ?**



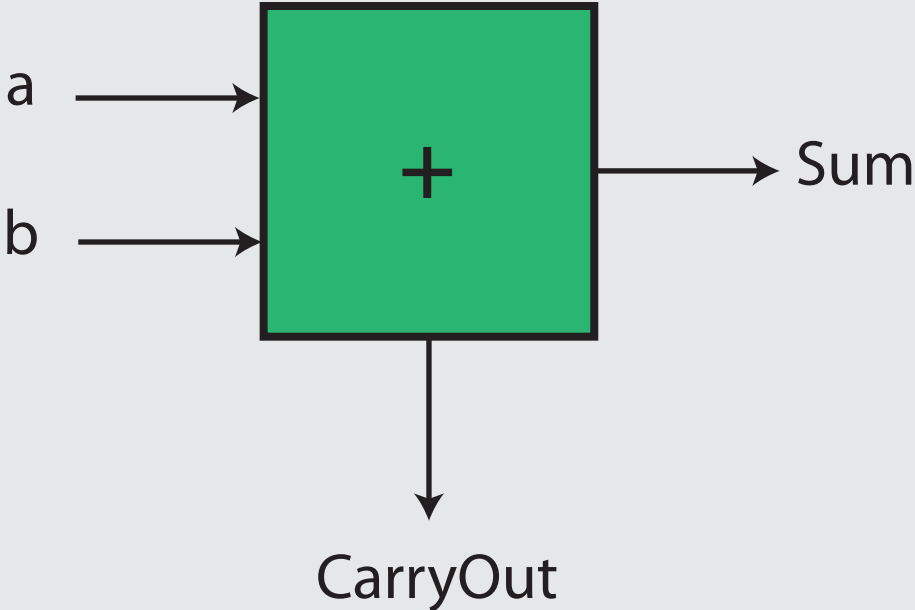
Can we implement this algorithm using logical gates ?



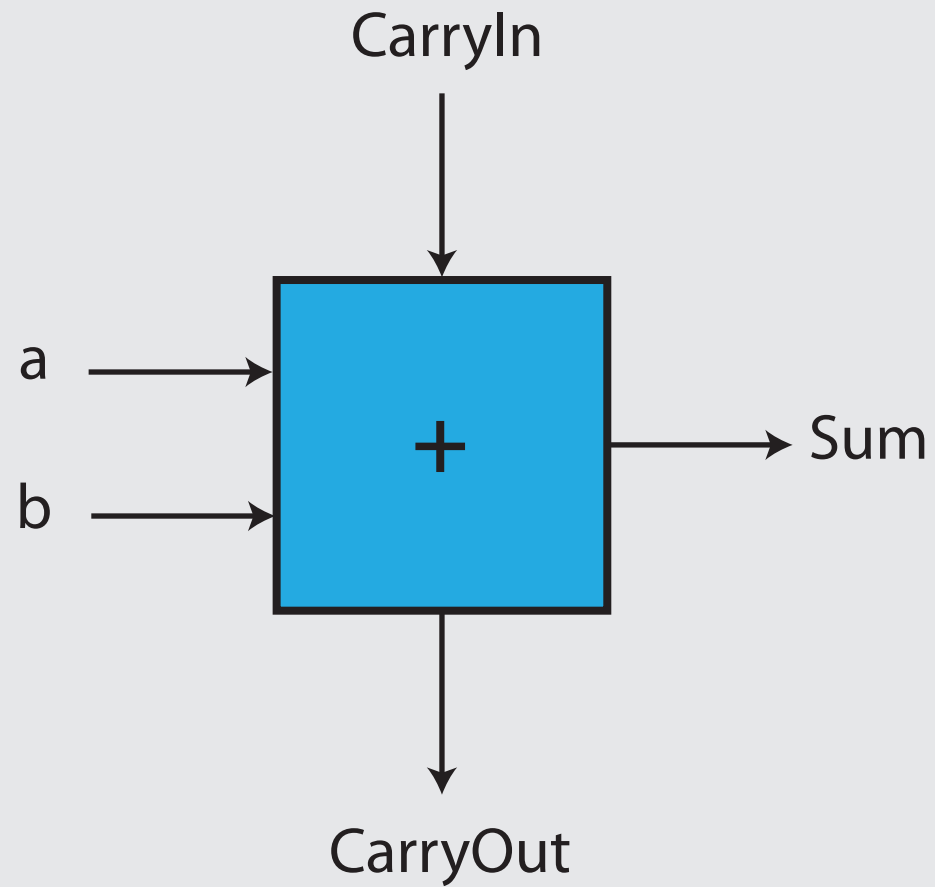
## Ripple Carry Adder



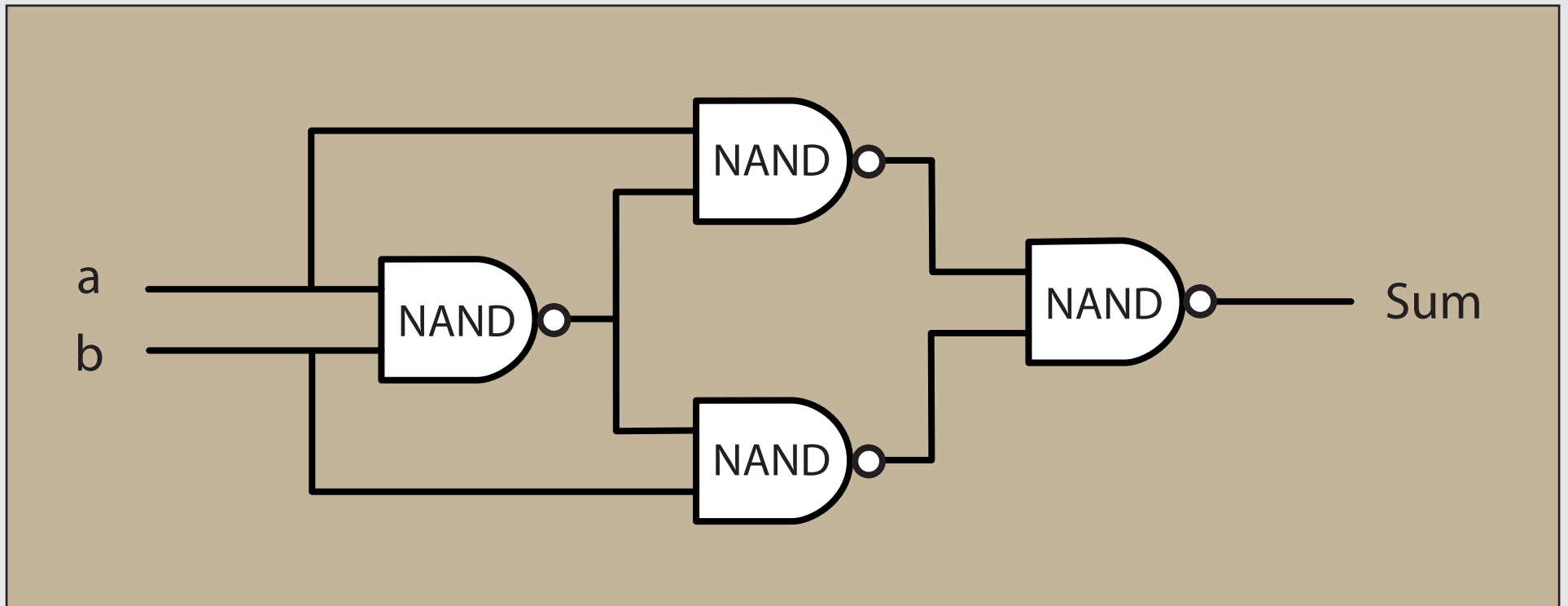
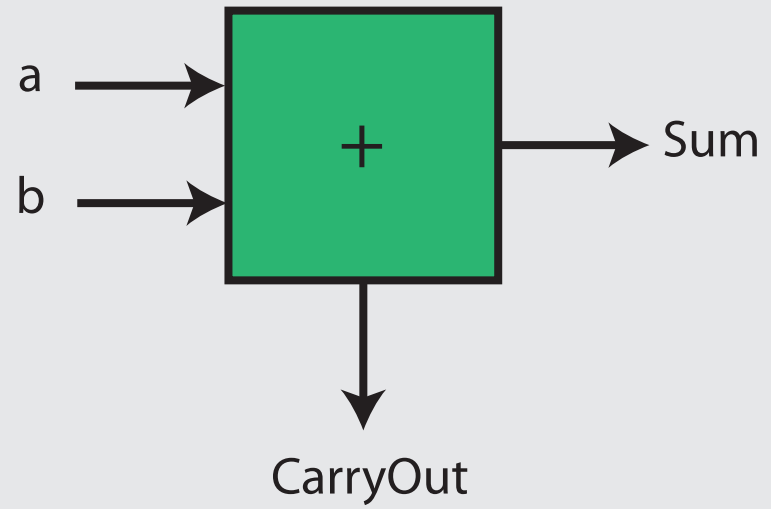
# Half adder



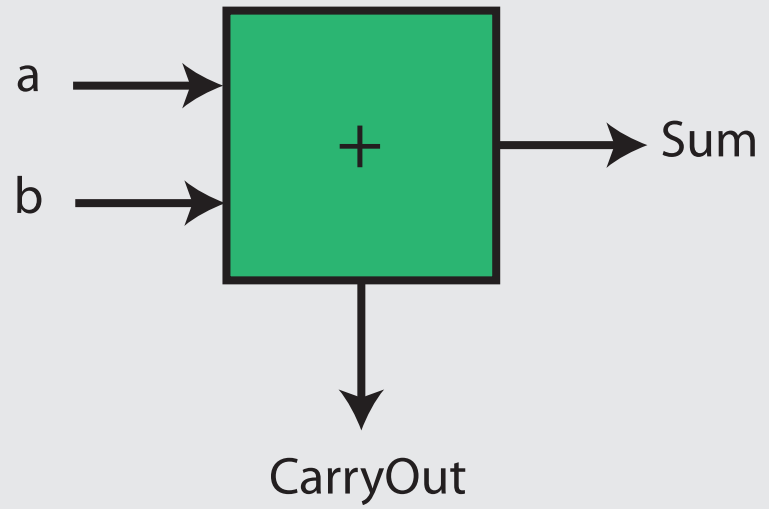
# Full adder



# Half adder

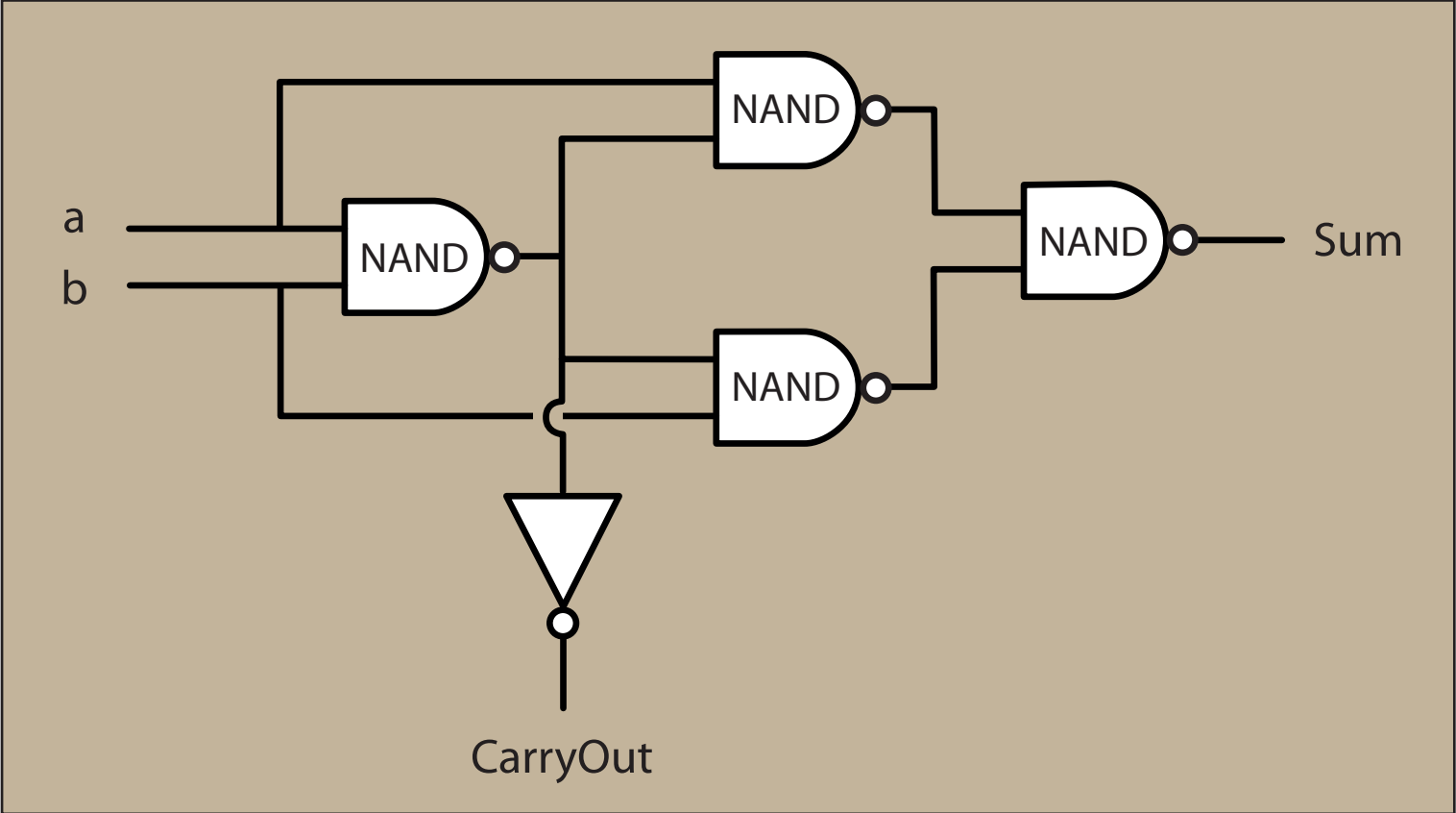
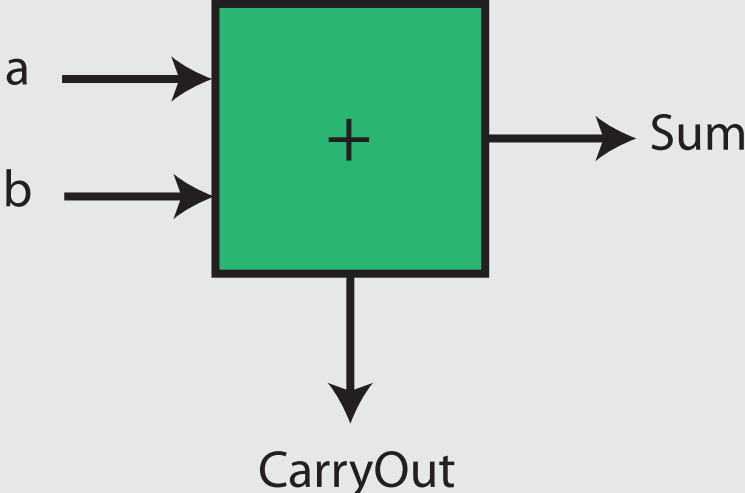


# Half adder

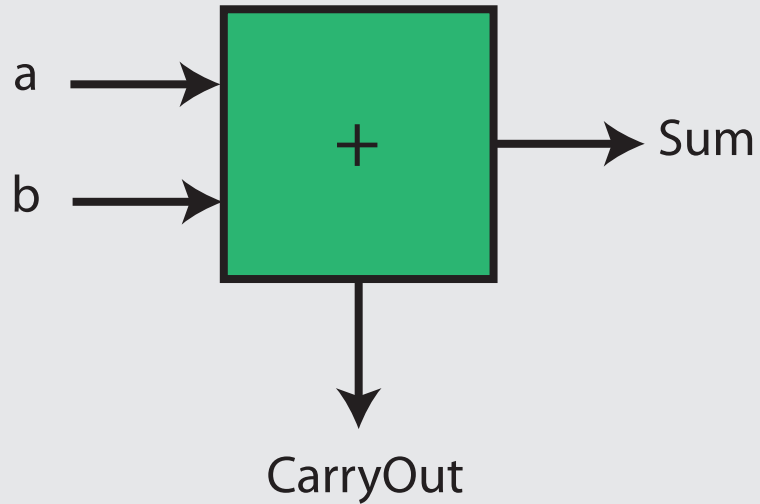




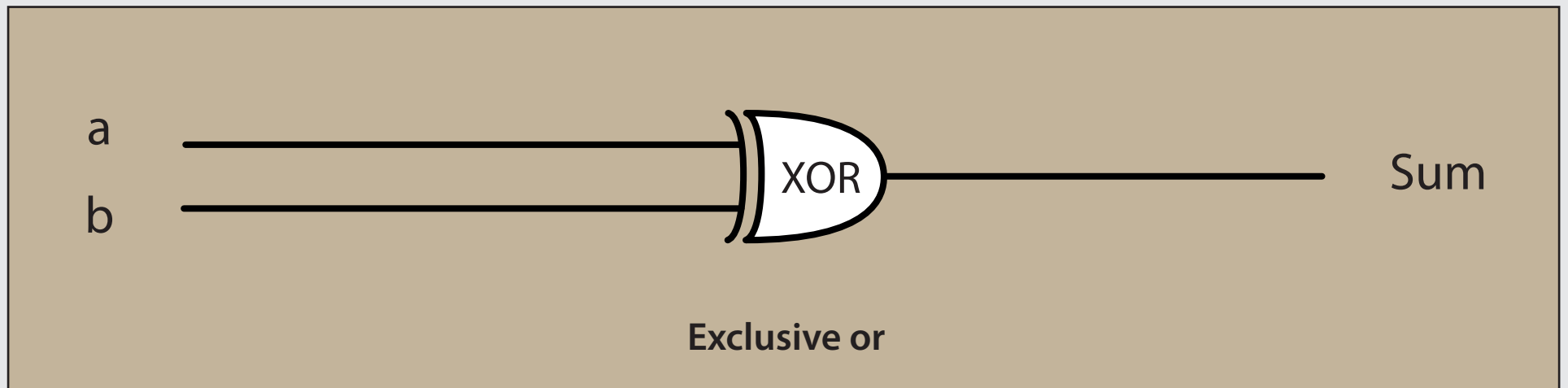
# Half adder



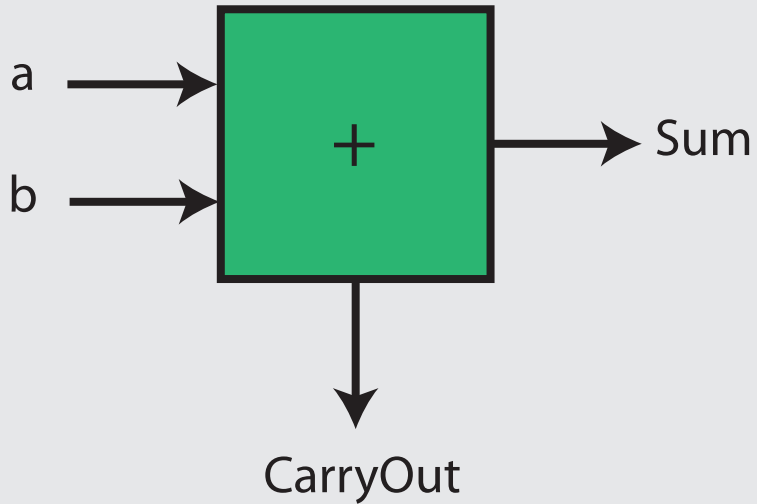
# Half adder



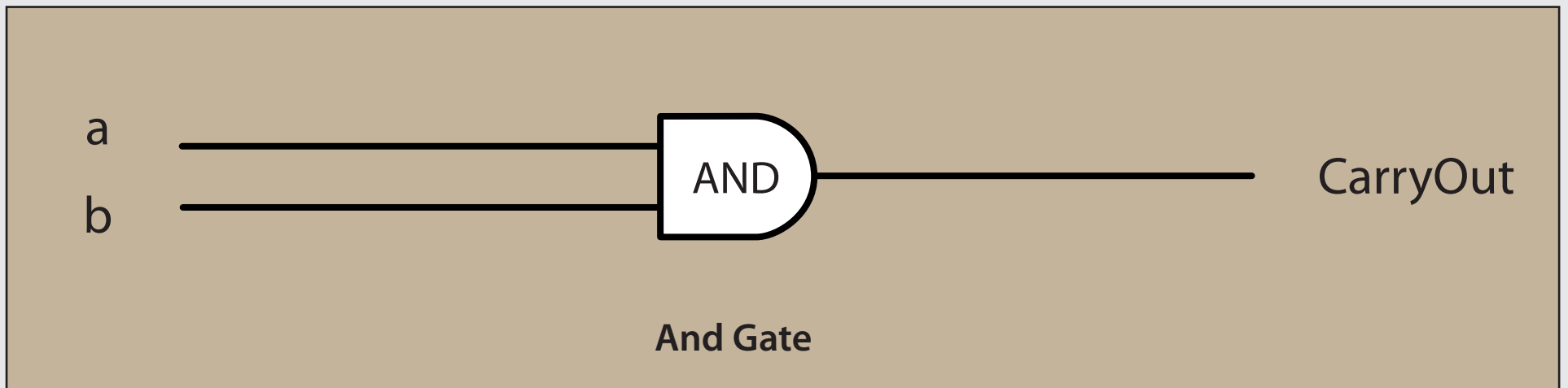
a	b	Sum
0	0	0
0	1	1
1	0	1
1	1	0



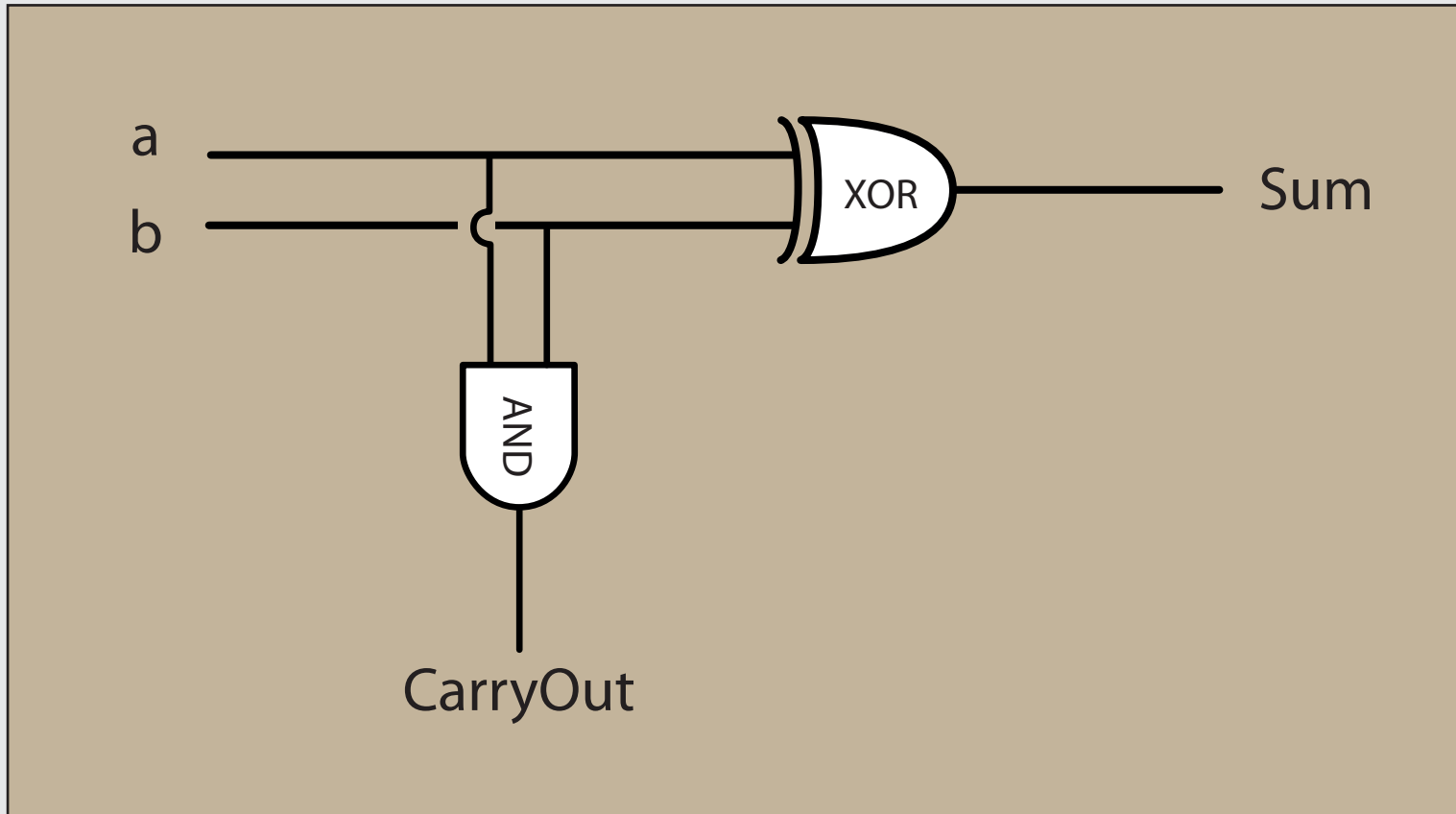
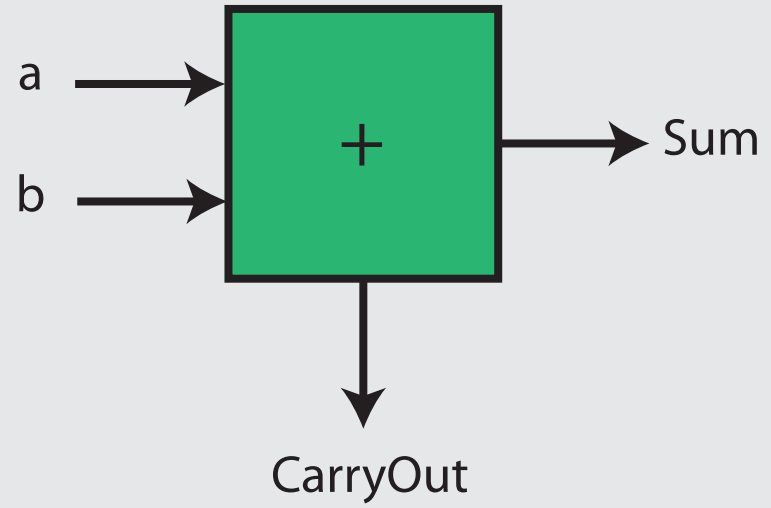
# Half adder



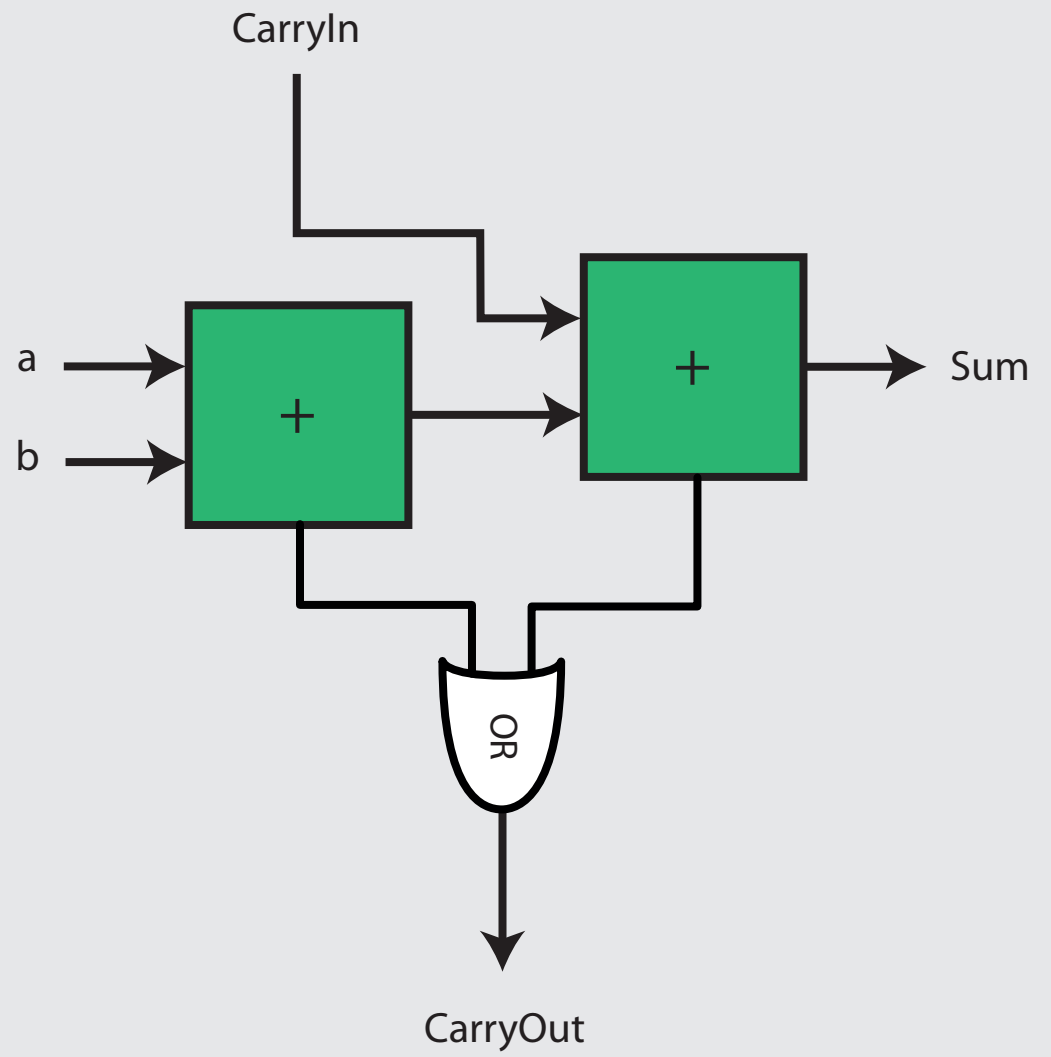
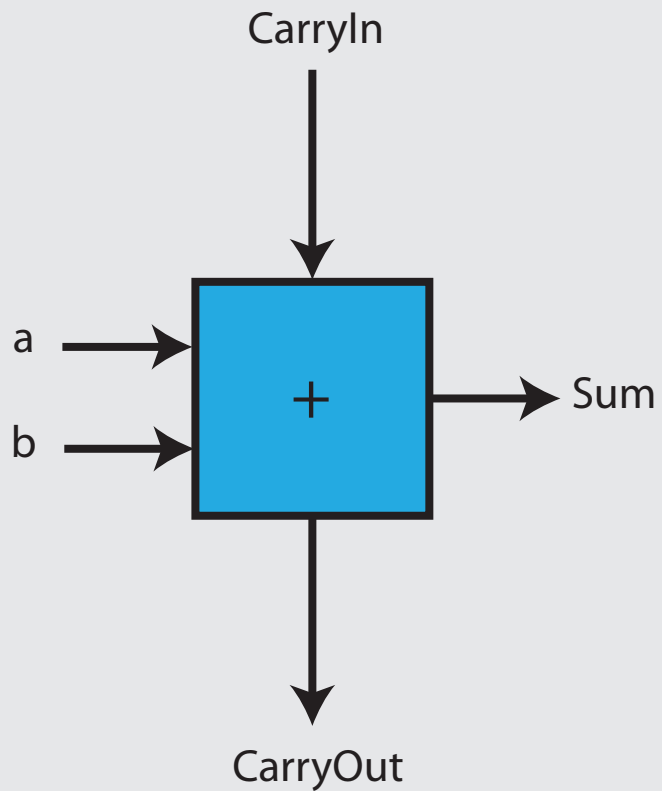
a	b	CarryOut
0	0	0
0	1	0
1	0	0
1	1	1



# Half adder



# Possible implementation of the full adder

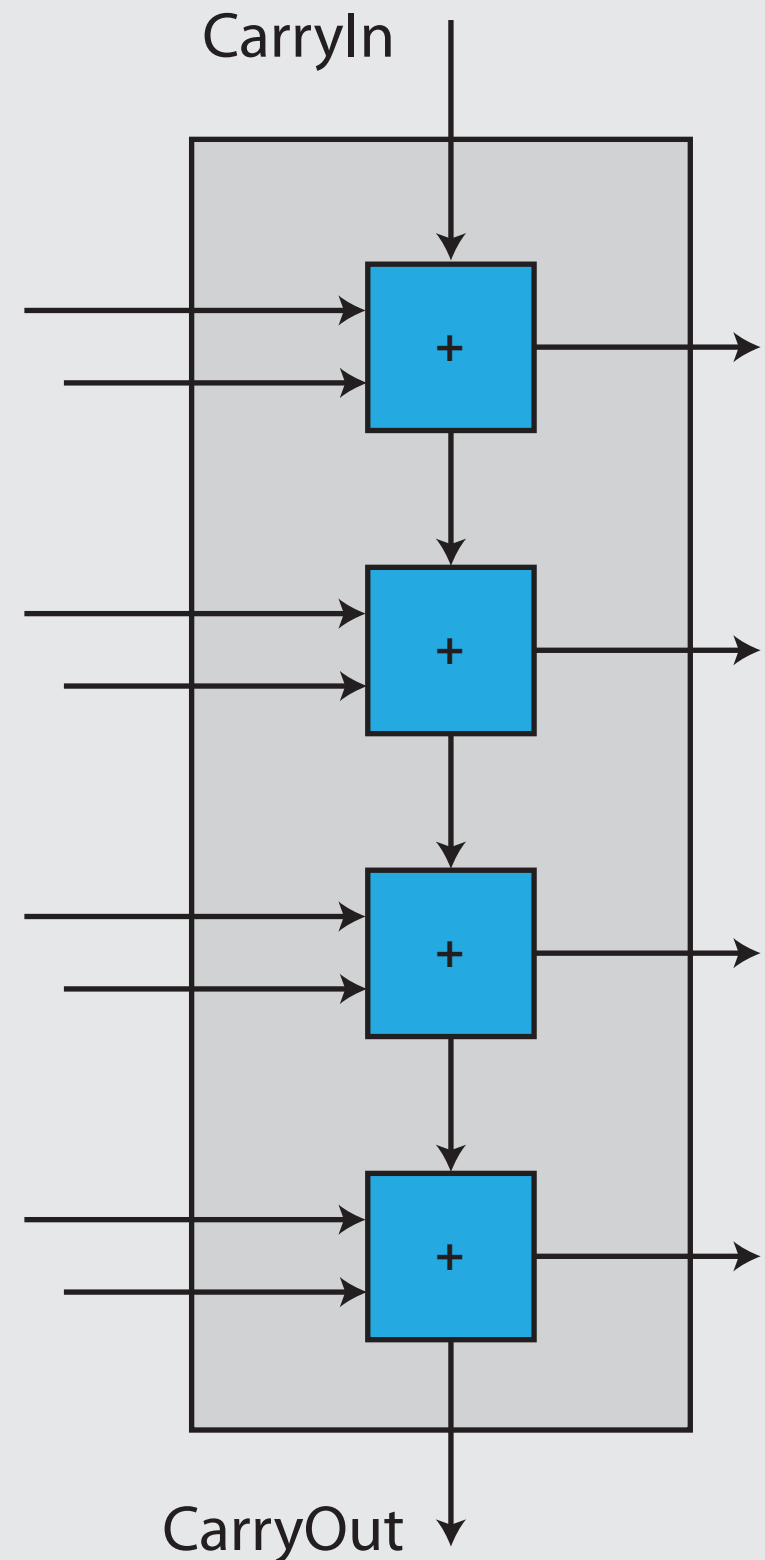


# Look Ahead Carry

Engineers have noticed that it is possible to compute the value of the CarryOut on four Full Adders ( or any similar number ) in a shorter amount of « time » than it takes to compute it through the four Full Adders.

Quite obviously, the value of the CarryOut depends on the values of the CarryIn as well as the values of the eight inputs.

This observation is useful because it enables one to « accelerate » the Ripple Carry Adder circuit computing on sixteen or thirty-two bits numbers into a faster Look Ahead Carry Adder.



# Signed numbers

# Representation of negative numbers

Three known solutions :

- one's complement : replace each bit 0 by a bit 1 and conversely
- two's complement : add 1 to the one's complement
- sign or magnitude : keep the binary number just as it is and turn the first bit 0 into 1

	<b>one's complement</b>	<b>two's complement</b>	<b>sign or magnitude</b>
<b>5</b>	<b>0000 0000 0000 0101</b>	<b>0000 0000 0000 0101</b>	<b>0000 0000 0000 0101</b>
<b>-5</b>	<b>1111 1111 1111 1010</b>	<b>1111 1111 1111 1011</b>	<b>1000 0000 0000 0101</b>



# One's complement

One main drawback of this convention is that there are two zero's :

- the positive zero :           0000 0000 0000 0000
- the negative zero :         1111 1111 1111 1111

In particular, when one applies that convention on 16-bit integers, and adds a positive number  $n$  with its negative counterpart  $-n$  one obtains :

1111 1111 1111 1111

This justifies the terminology « one's complement » .

# Two's complement

One main advantage of this convention is that it is compatible with addition on unsigned numbers.

In particular, when one applies this convention on 16-bit numbers, the sum of a positive number  $n$  with its negative counterpart  $-n$  is equal to zero :

0000 0000 0000 0000

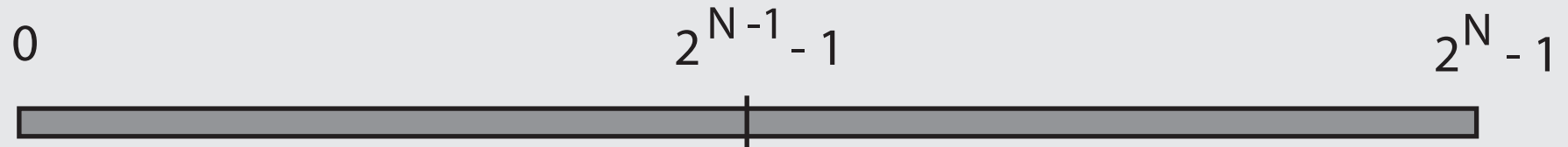
with carry 1.

Another way to look at it is to say that the sum is equal to :

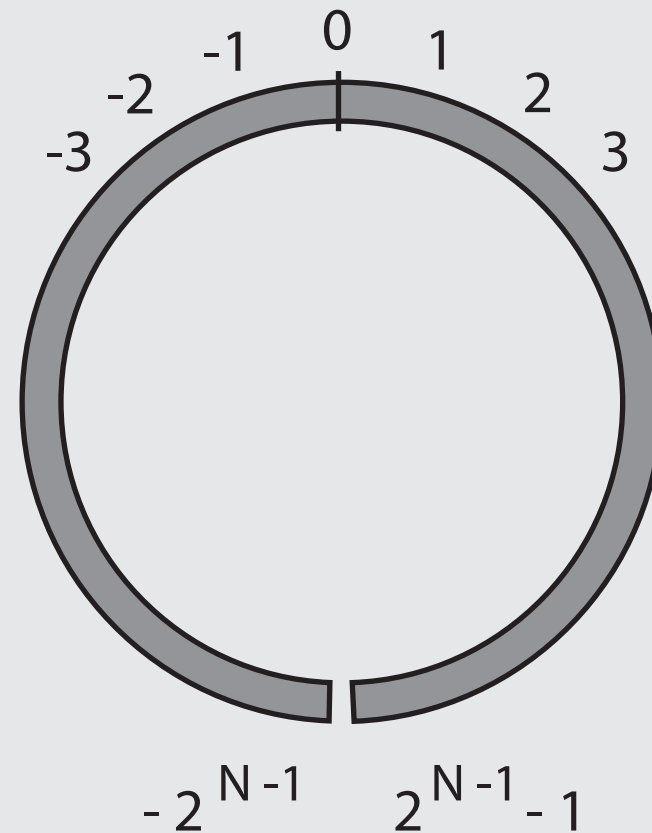
$$2_{\text{ten}}^{16} = 1\ 0000\ 0000\ 0000\ 0000_{\text{two}}$$

This justifies the terminology « two's complement » .

## Unsigned numbers



## Two's complement



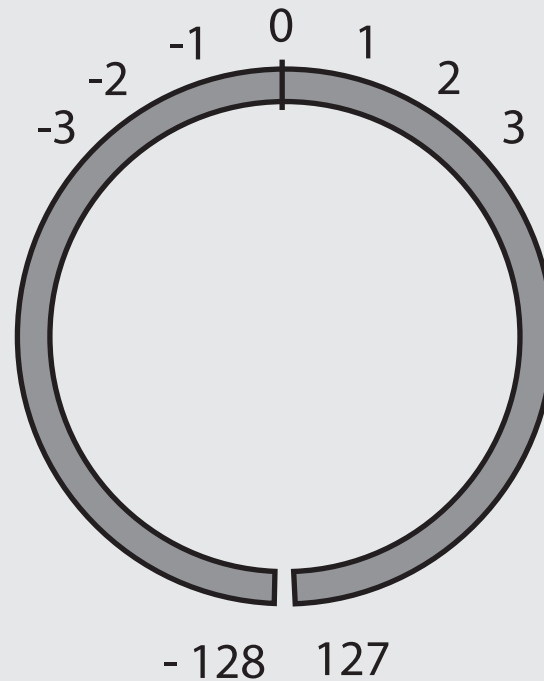
Here,  $N$  denotes the number of bits in the notation.

# Number of bits $N = 8$

Unsigned numbers



Signed numbers



# Number of bits $N = 8$

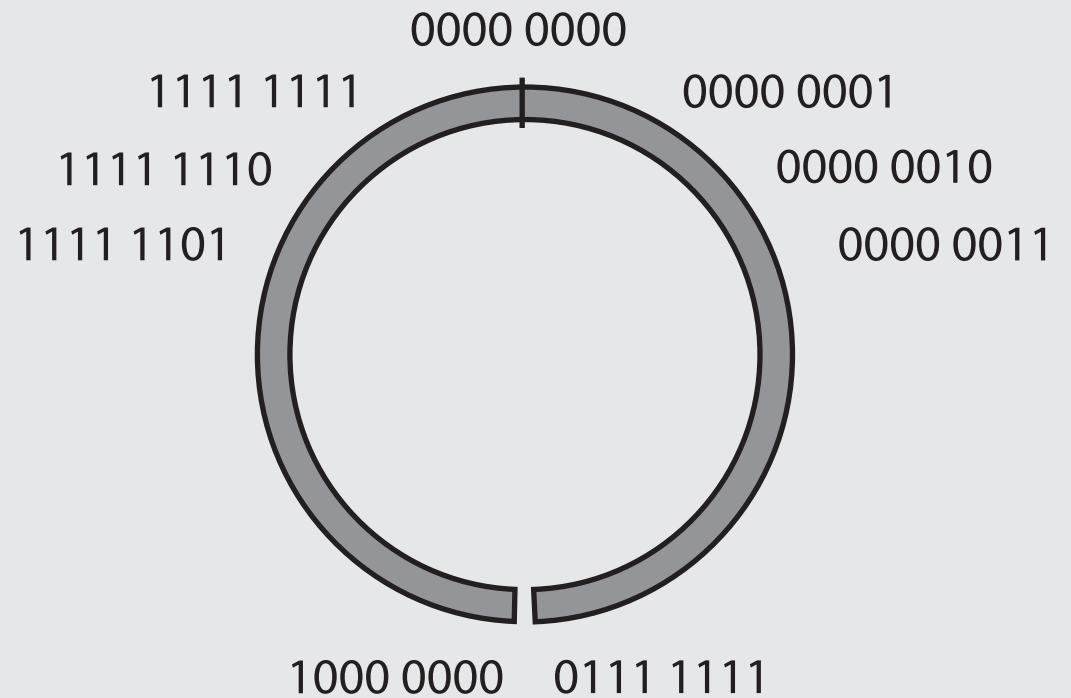
Unsigned numbers

0000 0000

1111 1111

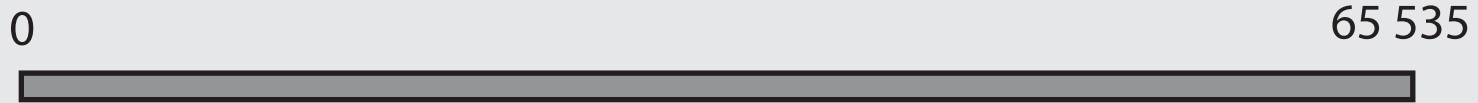


Signed numbers

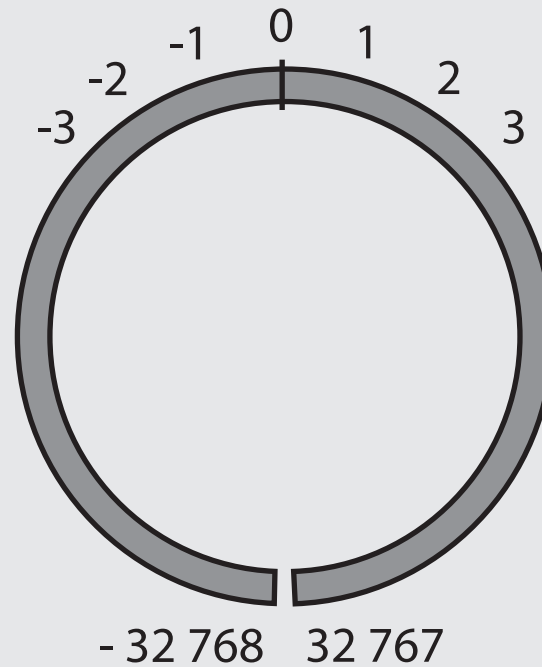


# Number of bits $N = 16$

Unsigned numbers

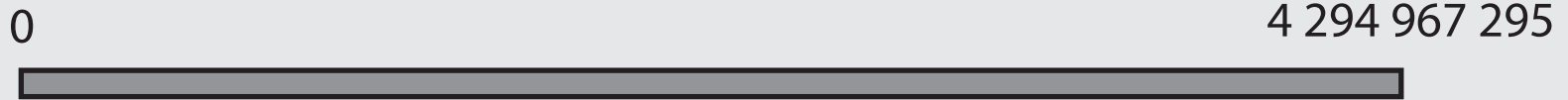


Signed numbers

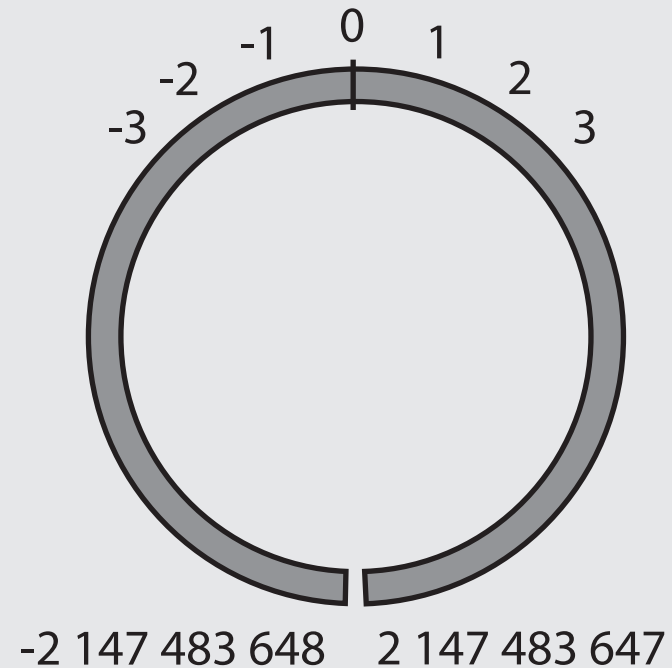


# Number of bits $N = 32$

Unsigned numbers



Signed numbers



# Two's complement

One main advantage of the notation is that it is sufficient to look at the most significant bit of a number to know whether that number is positive or negative :

- positive : when the most significant bit is equal to 0
- negative : when the most significant bit is equal to 1

Another main advantage is that it enables to do subtractions using the same algorithm as for addition :

$$\begin{array}{r} 0001\ 0001 \\ +\ 1111\ 1011 \\ \hline 0000\ 1100 \end{array}$$

$$\begin{array}{r} 17 \\ -\ 5 \\ \hline 12 \end{array}$$



# Overflow

When adding signed numbers, one should be careful about overflow :

- Adding a positive number with a negative number is always safe
- Adding two positive numbers induces an overflow precisely when their sum is larger than  $2^{N-1} - 1$
- Adding two negative numbers induces an overflow precisely when their sum ( in absolute value ) is larger than  $2^{N-1}$

One simple way to check that there has been an overflow :

1. No overflow when the signs of the two operands are different
2. when the signs of the two operands are the same :
  - 2a. no overflow when the sign of their sum is the same
  - 2b. overflow when the sign of their sum is different

# Overflow

When adding signed numbers, one should be careful about overflow :

- Adding a positive number with a negative number is always safe
- Adding two positive numbers induces an overflow precisely when their sum is larger than  $2^{N-1} - 1$
- Adding two negative numbers induces an overflow precisely when their sum ( in absolute value ) is larger than  $2^{N-1}$

Another way to check whether there has been an overflow :

An overflow occurs during an addition between signed numbers precisely when the carry **into** the most significant bit and the carry **from** the most significant bit are **different**.

# Implementing add with addu in MIPS

```
#addition with overflow
```

```
addu $t0, $t1, $t2      # does not interrupt in case of overflow!  
xor  $t3, $t1, $t2      # if signs of $t1 and $t2 differ then $t3 < 0  
slt  $t3, $t3, $zero    # compare the value of $t3 with zero  
bne  $t3, $zero, No_overflow  
                                # no overflow when signs differ  
  
xor  $t3, $t0, $t1      # when signs of $t1 and $t2 match  
                                # compare the sign of $t1  
                                # with the sign of the sum $t0  
                                # if signs of $t0 and $t1 differ then $t3 < 0  
slt  $t3, $t3, $zero    # compare the value of $t3 with zero  
bne  $t3, $zero, Overflow  
                                # jump to Overflow when sign differ
```

A simple procedure in MIPS to check the presence of an overflow during an addition

# Illustration of overflow

For simplicity, let us suppose that  $N = 8$ .

In that case :

$$2^N = 256$$

$$\begin{array}{r} 0101\ 1001 \\ +\ 0100\ 1010 \\ \hline 1010\ 0011 \end{array}$$

$$\begin{array}{r} 1010\ 1010 \\ +\ 1100\ 1100 \\ \hline 0111\ 0110 \end{array}$$

$$\begin{array}{r} 89 \\ +\ 74 \\ \hline 163 = -93 \end{array}$$

$$\begin{array}{r} -86 \\ +\ -52 \\ \hline -138 = 118 \end{array}$$

# Signed vs. unsigned addition in MIPS

There are two instructions for addition of registers in MIPS and this may be confusing...

addu : does the expected thing

add : throws an exception in case of an overflow

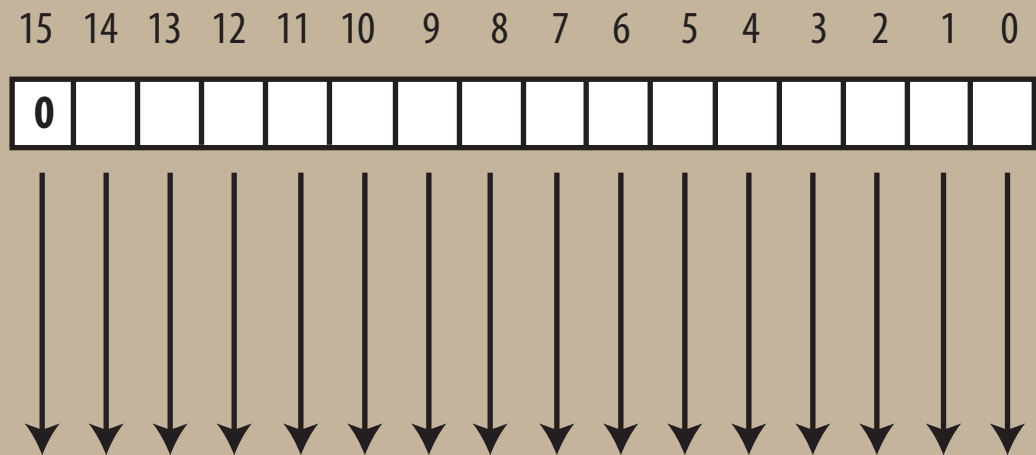
What is an exception ( also called an interrupt ) ?

It is an unscheduled event which disrupts the program execution. The processor saves the address of the offending instruction in the **exception program counter** ( EPC ) and then jumps to an address specified by the operating system. After performing whatever action is required because of the exception, the operating system can terminate the program or continue using the EPC to determine where to restart the execution of the program.

# Sign extension

This is an important operation, which should be well-understood ...

Sign extension for positive numbers :

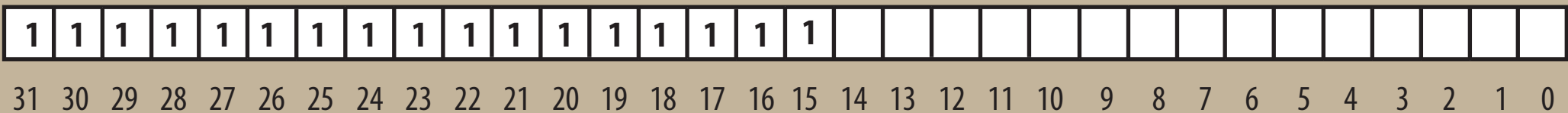
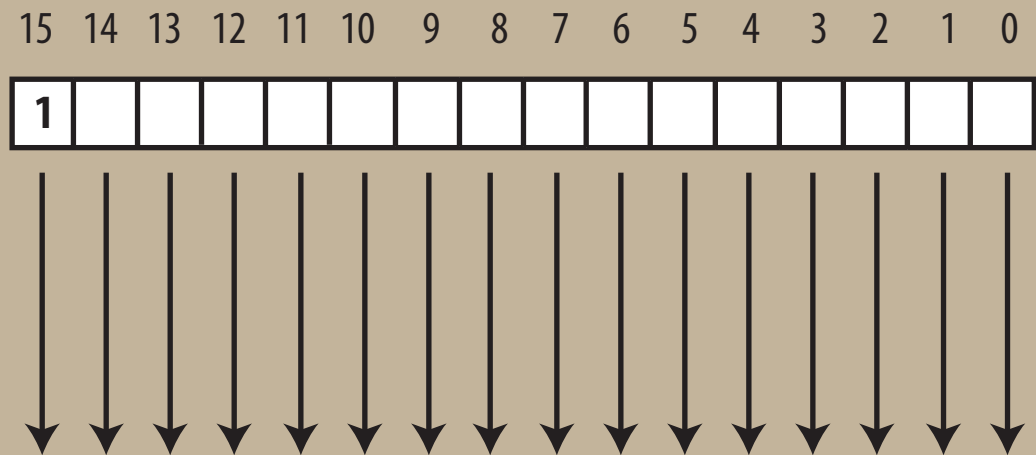


Here we sign extend 16 bit numbers into 32 bit numbers

# Sign extension

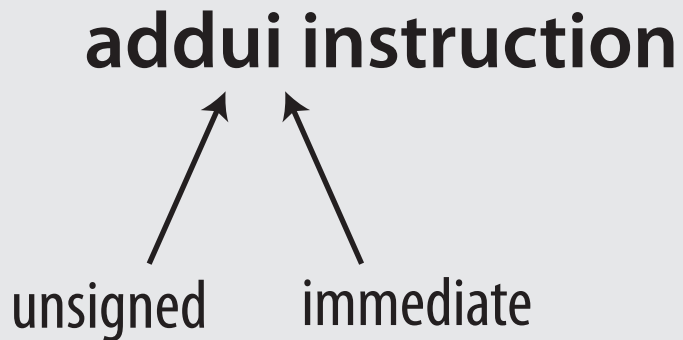
This is an important operation, which should be well-understood ...

Sign extension for negative numbers :

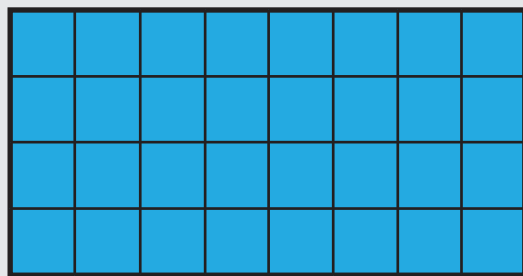


Here we sign extend 16 bit numbers into 32 bit numbers

# In the MIPS instruction set ...

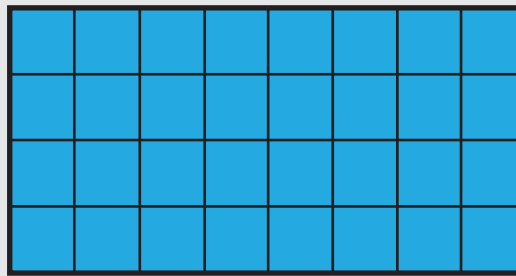


0	0	1	0	0	0	rs	rs
rs	rs	rs	rt	rt	rt	rt	rt
b <sub>15</sub>	b <sub>14</sub>	b <sub>13</sub>	b <sub>12</sub>	b <sub>11</sub>	b <sub>10</sub>	b <sub>9</sub>	b <sub>8</sub>
b <sub>7</sub>	b <sub>6</sub>	b <sub>5</sub>	b <sub>4</sub>	b <sub>3</sub>	b <sub>2</sub>	b <sub>1</sub>	b <sub>0</sub>



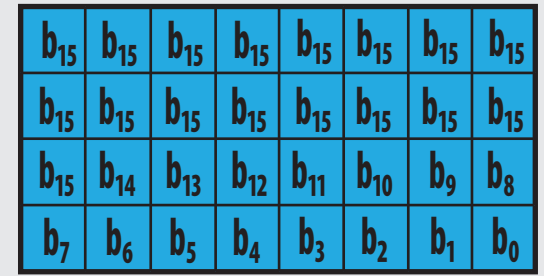
Register [ rt ]

=



Register [ rs ]

+



SignExtImm

Here, unsigned addition means that no exception will be launched in case of overflow



# Multiplication

# Multiplication

Multiplicand		1 0 0 0	ten
Multiplier	x	1 0 0 1	ten
		<hr/>	
		1 0 0 0	
		0 0 0 0	
		0 0 0 0	
		1 0 0 0	
		<hr/>	
Product		1 0 0 1 0 0 0	ten

# The multiplication algorithm

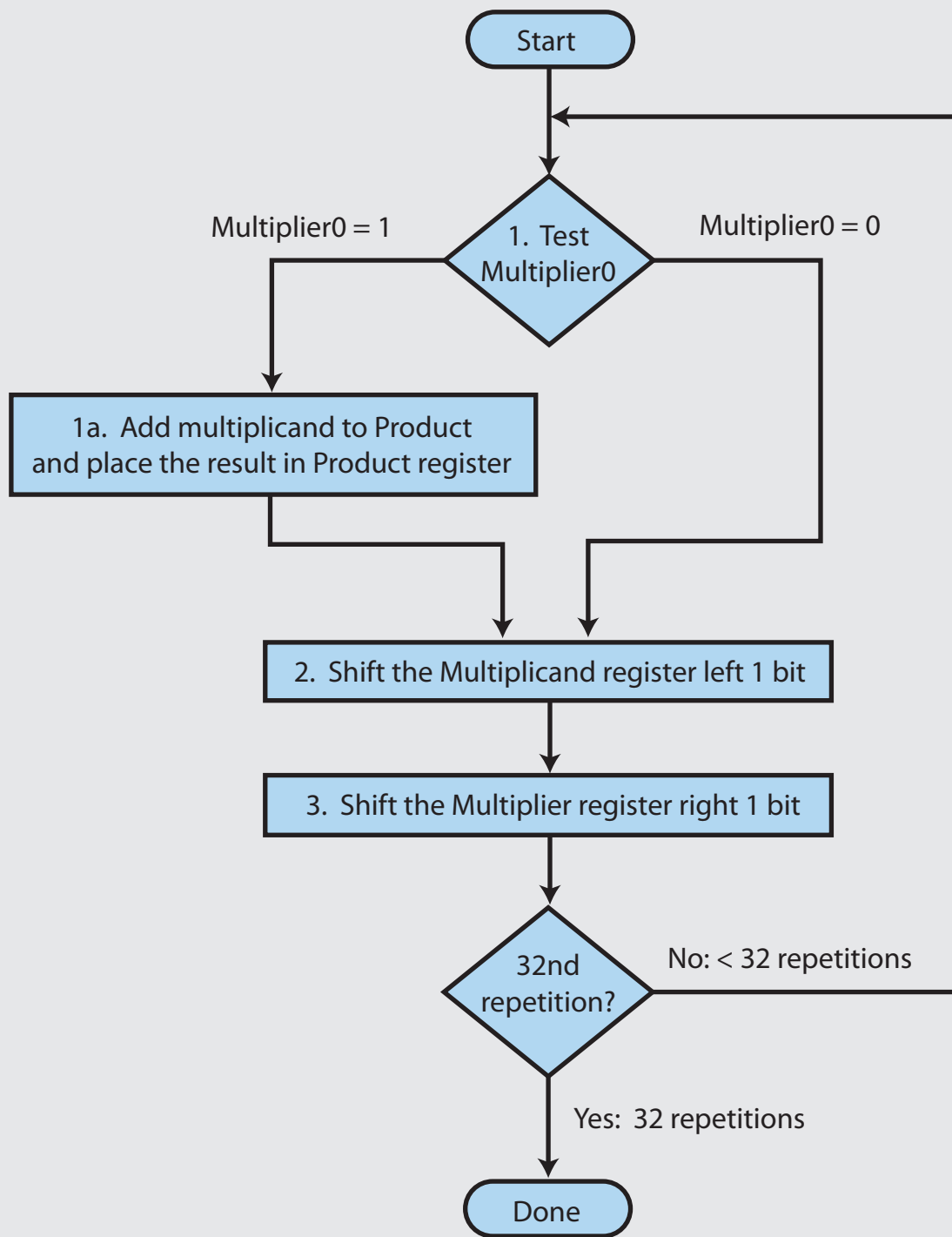


Figure 3.4 in  
Patterson and Hennessy

# Floating point

# Normalized numbers

In many situations, one wants to represent numbers like :

The number  $\pi$                        $\pi = 3.1515926535897932 \dots$

the Euler number                       $e = 2.718281828459045 \dots$

or the Planck constant               $\hbar = 1.054571726 \times 10^{-34} \text{ J.s}$

To that purpose, one uses floating points.

Every real number can be normalized as follows :

$$\pm x.yyyyyyyy\dots \times 10^{\pm zzzzzzz}$$

where  $x$  is a natural number different from 0, and  $y$ 's and  $z$ 's are numbers.

Typically,  $1.0 \times 10^{-3}$  is normalized while 0.001 is not normalized.

# Powers of 10

$10^{15}$	:	peta
$10^{12}$	:	tera
$10^9$	:	giga
$10^6$	:	mega
$10^3$	:	kilo
$10^{-3}$	:	milli
$10^{-6}$	:	micro
$10^{-9}$	:	nano
$10^{-12}$	:	pico
$10^{-15}$	:	femto

Sunway TaihuLight

Illustration: my 神威·太湖之光 can compute up to 93 petaflops

# Normalized binary numbers

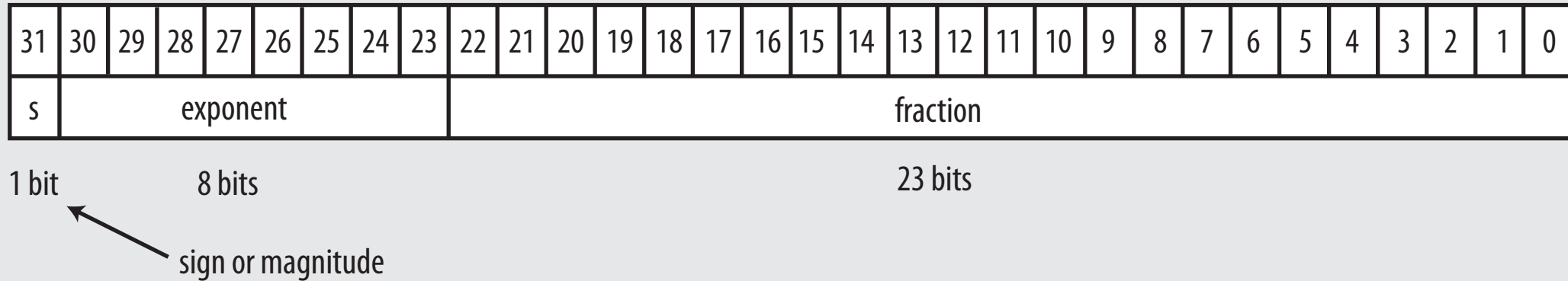
Similarly, every real number can be normalized as follows :

$$\pm 1.xxxxxxxxxx\dots \times 2^{\pm yyyyyyy}$$

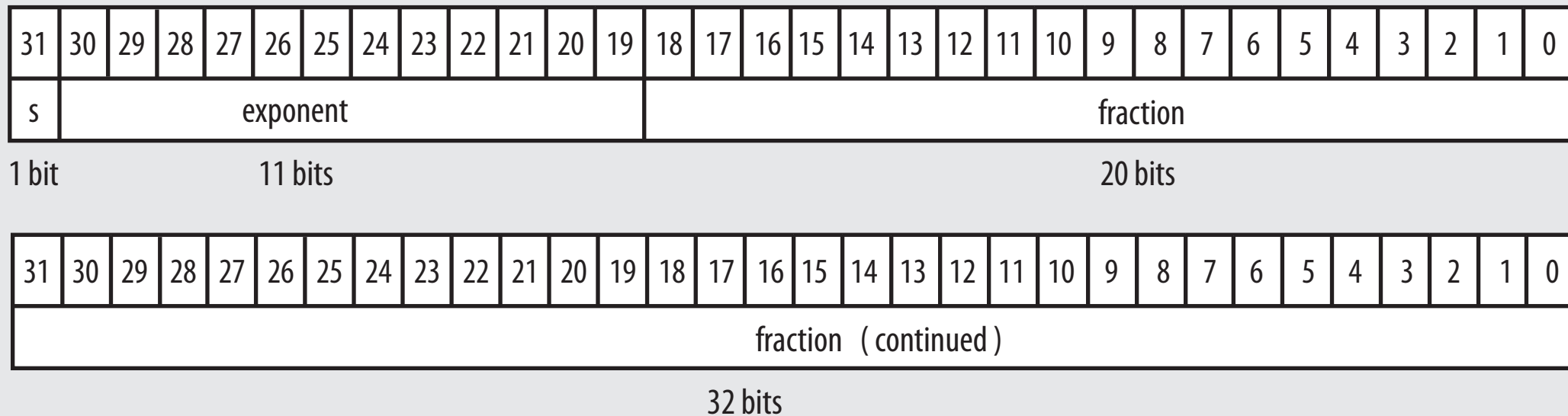
This normalized notation provides a handy way to manipulate real numbers, at least in an approximative way.

# Binary representation of floating points

## float ( single precision )



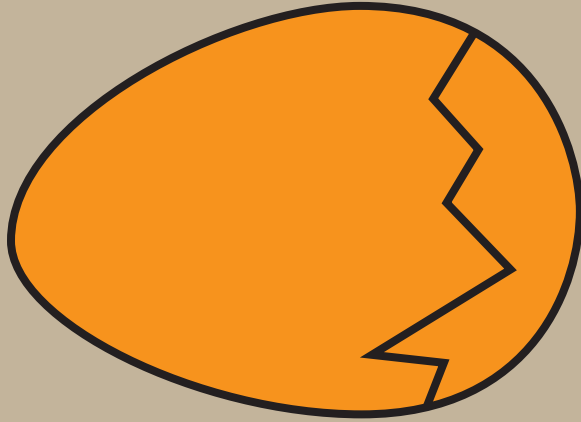
## double ( double precision )





# **The issue of Big endian vs. Little endian**

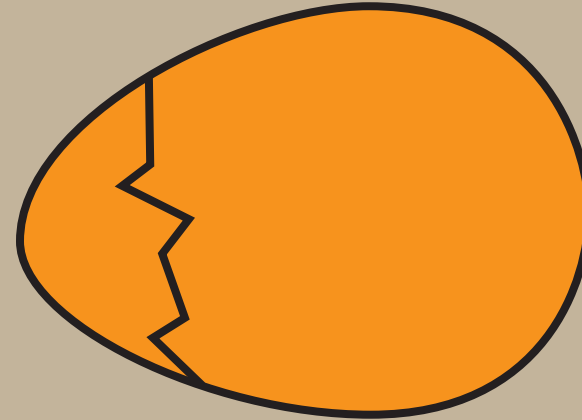
# Big endian vs. Little endian



BIG ENDIAN :

The way people always broke their eggs in the Lilliput land

A quite common convention



LITTLE ENDIAN :

The way the king then ordered the people to break their egg

Intel x86 family for instance

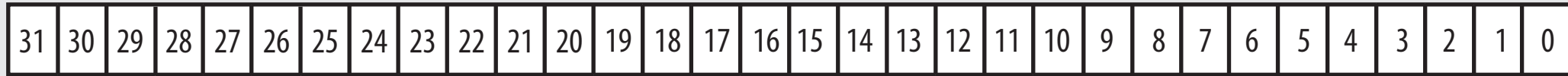
Gulliver's travels by Jonathan Swift ( 1726, amended 1735 )

A paper by Danny Cohen ( published 1 April 1980 )

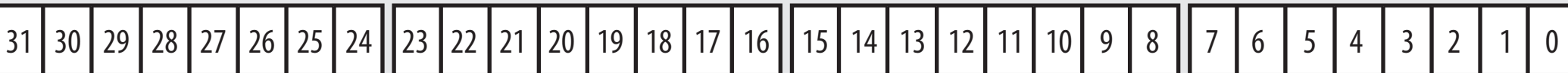
See also : The History of the Microcomputer - Invention and evolution by Stanley Mazor, Dec. 1995

# Words in memory

**word**



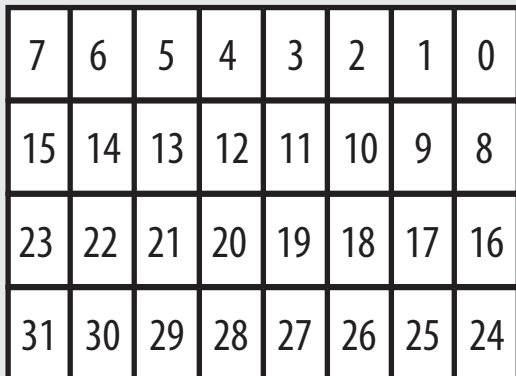
**word = four bytes**



most significant byte

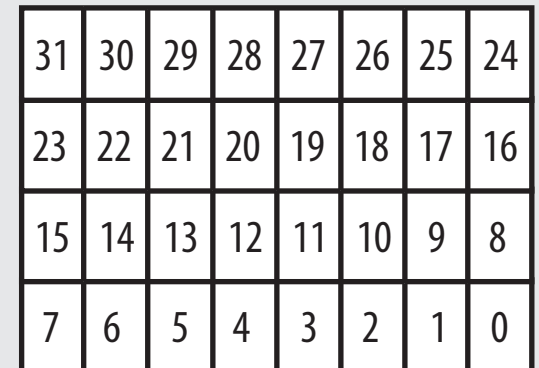
least significant byte

**big endian**



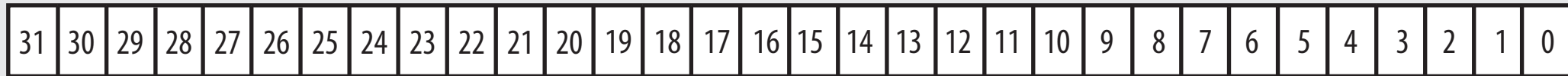
memory address increasing

**little endian**

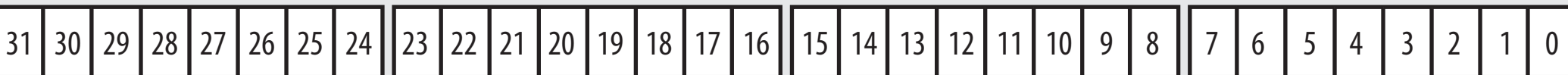


# Words in memory

word



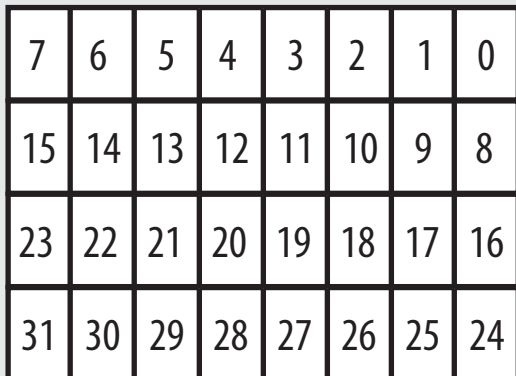
word = four bytes



most significant byte

least significant byte

big endian



byte 3

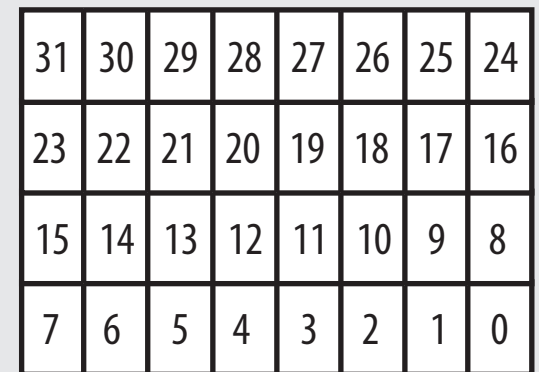
byte 2

byte 1

byte 0



little endian



byte 3

byte 2

byte 1

byte 0