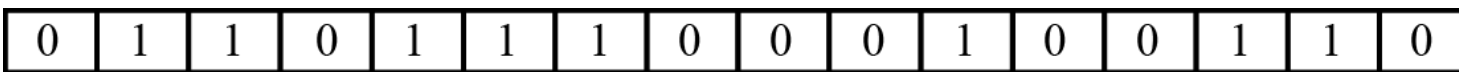


STORING REAL NUMBERS

A **number** is changed to binary before being stored in computer memory, as described earlier. There are two issues that need to be handled:

1. How to store the sign of the number (**we already know this**).
2. How to show the (radix) point.

For the (radix) point, computers use two different representations: **fixed-point** and **floating-point**. **The first is used to store a number as an integer, without a fraction part.** The second is used to store a number as a real number, with a fractional part.



Memory location

Decimal point / Radix point
(assumed position)

Fixed point representation of integers

An integer is normally stored in memory using fixed-point representation.

Applications of unsigned integers:

Counting- Addressing- storing other data types (text, images, audio and video)

Storing real numbers Continued

A real number is a number with an integral part and a fractional part. For example, 23.7 is a real number—the integral part is 23 and the fractional part is $7/10$. Although a fixed-point representation can be used to represent a real number, the result may not be accurate or it may not have the required precision. The next two examples explain why.

Real numbers with very large integral parts or very small fractional parts should not be stored in fixed-point representation.

Example 1:

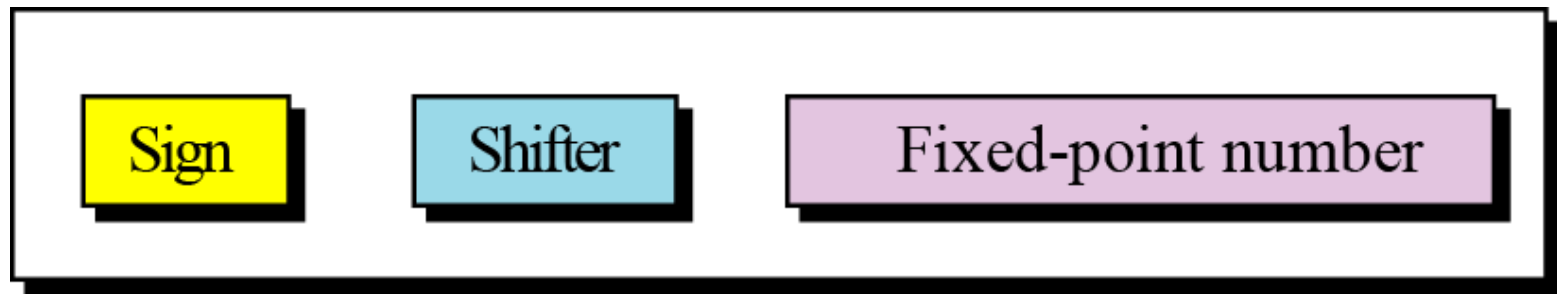
In the decimal system, assume that we use a fixed-point representation with **two digits to the right** of the decimal point and **fourteen digits to the left** of the decimal point, for a total of sixteen digits. The precision of a real number in this system is lost if we try to represent a decimal number such as 1.00234: the system stores the number as 1.00

Example 2:

In the decimal system, assume that we use a fixed-point representation with **six digits to the right** of the decimal point and **ten digits to the left** of the decimal point, for a total of sixteen digits. The accuracy of a real number in this system is lost if we try to represent a decimal number such as 236154302345.00. The system stores the number as 6154302345.00. The integral part is much smaller than it should be.

Floating-point representation

The solution for maintaining accuracy or precision is to use **floating-point representation**.



Floating-point representation

A floating point representation of a number is made up of three parts: a sign, a shifter and a fixed-point number.

Floating-point representation is used in science to represent very small or very large decimal numbers. In this representation called **scientific notation**, the fixed-point section has only one digit to the left of point and the shifter is the power of 10.

Example:

The following shows the decimal number

7,425,000,000,000,000,000,000.00

in scientific notation (floating-point representation).

Actual number	→	+	7,425,000,000,000,000,000,000.00
Scientific notation	→	+	7.425×10^{21}

The three sections are the **sign** (+), the **shifter** (21) and the **fixed-point part** (7.425). Note that the shifter is the exponent.

Some programming languages and calculators shows the number as +7.425E21

Example:

Show the number -0.00000000000000232

in scientific notation (floating-point representation).

Solution

We use the same approach as in the previous example—we move the decimal point after the digit 2, as shown below:

Actual number	→	–	0.00000000000000232
Scientific notation	→	–	2.32×10^{-14}

The three sections are the **sign** (–), the **shifter** (-14) and the **fixed-point part** (2.32). Note that the shifter is the exponent.

Example:

Show the number,

$$(101001000000000000000000000000000000.00)_2$$

in floating-point representation.

Solution

We use the same idea, keeping only one digit to the left of the radix (decimal) point.

Actual number $\rightarrow + (101001000000000000000000000000000000.00)_2$

Scientific notation $\rightarrow + 1.01001 \times 2^{32}$

Example:

Show the number

$$-(0.\underbrace{0000000000000000000000}_{\text{zeros}}101)_2$$

in floating-point representation.

Solution

We use the same idea, keeping only one digit to the left of the decimal point.

[illegible]

Scientific notation $\rightarrow -1.01 \times 2^{-24}$

Normalization

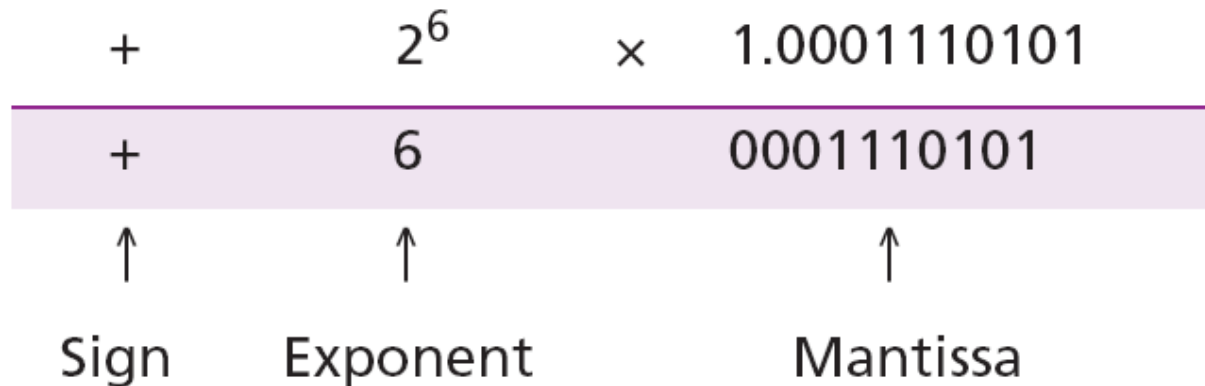
To make the **fixed part** of the representation **uniform**, both the scientific method (for the decimal system) and the floating-point method (for the binary system) **use only one non-zero digit on the left of the decimal point**. This is called **normalization**. In the decimal system this digit can be 1 to 9, while in the binary system it can only be 1. In the following, d is a non-zero digit, x is a digit, and y is either 0 or 1.

Decimal	→	±	d.xxxxxxxxxxxxxxx
Binary	→	±	1.yyyyyyyyyyyyyyy

Note: d is 1 to 9 and each x is 0 to 9

Note: each y is 0 or 1

To store 1000111.0101_2 in memory, using floating point representation; First we put it in normalized form 1.0001110101×2^6 , and then store it as shown below



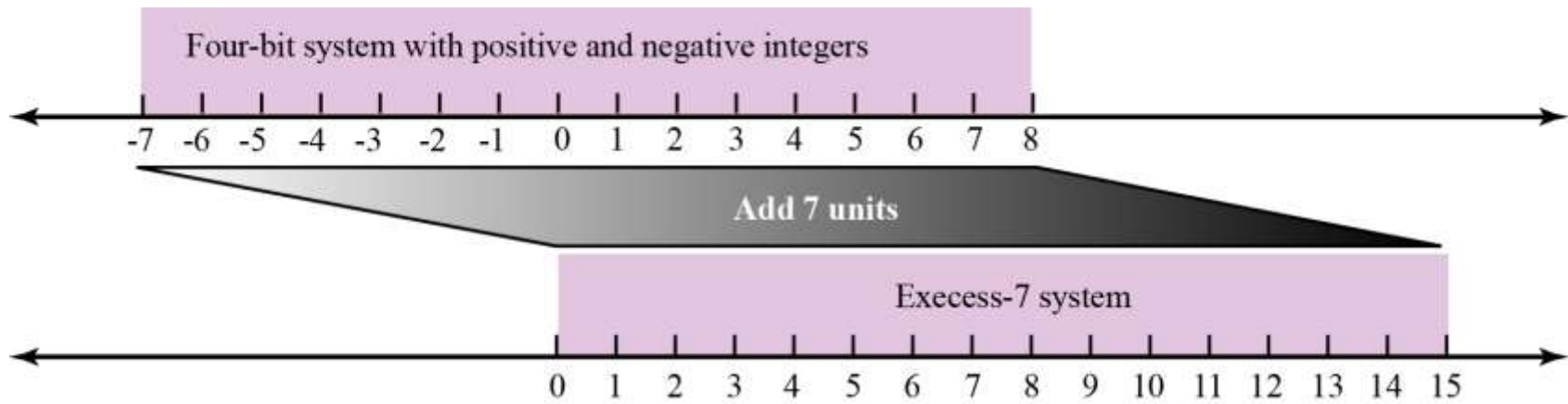
Note that the point and the bit 1 to the left of the fixed-point section (Mantissa) are not stored; They are implicit (hidden or not shown).

Excess Notation

- The **exponent**, the power that shows how many bits the decimal point should be moved to the left or right, is a signed number.
- Although this could have been stored using **two's complement representation**, a new representation, called the **Excess notation**, is used instead.
- In the Excess notation, **both positive and negative integers are stored as unsigned integers**.
- To represent a positive or negative integer, a positive integer (called a bias) is added to each number to shift them uniformly to the non-negative side. The value of this bias is $2^{m-1} - 1$, where m is the size of the memory to store the exponent.

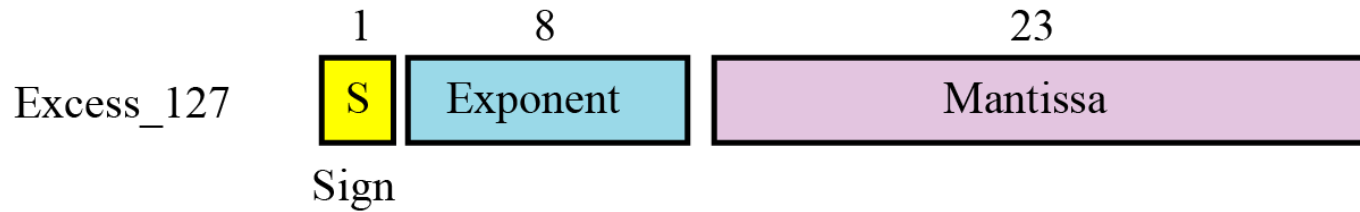
Example:

We can express **sixteen integers** in a number system with **4-bit allocation**. By adding seven units to each integer in this range, we can uniformly translate all integers to the right and make all of them positive without changing the relative position of the integers with respect to each other, as shown in the figure. The new system is referred to as **Excess-7**, or biased representation with biasing value of 7.

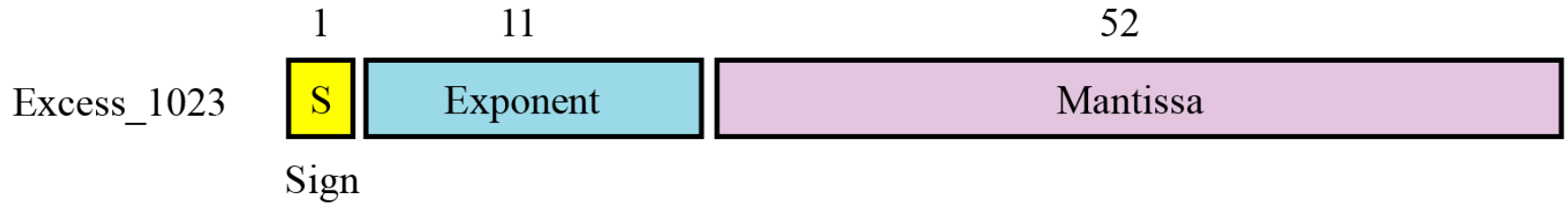


Shifting in Excess representation

IEEE Standard



a. Single precision (32 bits)



b. Double precision (64 bits)

IEEE standards for floating-point representation

IEEE Specifications

Specifications of the two IEEE floating-point standards

<i>Parameter</i>	<i>Single Precision</i>	<i>Double Precision</i>
Memory location size (number of bits)	32	64
Sign size (number of bits)	1	1
Exponent size (number of bits)	8	11
Mantissa size (number of bits)	23	52
Bias (integer)	127	1023

Storage of IEEE standard floating point numbers:

1. Store the sign in S (0 or 1).
2. Change the number to binary.
3. Normalize.
4. Find the values of E and M.
5. Concatenate S, E, and M.

Example 1:

Show the Excess_127 (single precision) representation of the decimal number 5.75

Solution

- The sign is positive, so $S = 0$.
- Decimal to binary transformation: $5.75 = (101.11)_2$.
- Normalization: $(101.11)_2 = (1.0111)_2 \times 2^2$.
- $E = 2 + 127 = 129 = (10000001)_2$, $M = 0111$. We need to add nineteen zeros at the right of M to make it 23 bits.
- The representation is shown below:

0	10000001	011100000000000000000000
S	E	M

The number is stored in the computer as

01000000101110000000000000000000

Example 2:

Show the Excess_127 (single precision) representation of the decimal number -161.875

Solution

- The sign is negative, so $S = 1$.
- Decimal to binary transformation: $161.875 = (10100001.111)_2$.
- Normalization: $(10100001.111)_2 = (1.0100001111)_2 \times 2^7$.
- $E = 7 + 127 = 134 = (10000110)_2$ and $M = (0100001111)_2$.
- Representation:

1	10000110	010000111100000000000000
S	E	M

The number is stored in the computer as

11000011001000011110000000000000

Example 3:

Show the Excess_127 (single precision) representation of the decimal number -0.0234375

Solution

- $S = 1$ (the number is negative).
- Decimal to binary transformation: $0.0234375 = (0.0000011)_2$.
- Normalization: $(0.0000011)_2 = (1.1)_2 \times 2^{-6}$.
- $E = -6 + 127 = 121 = (01111001)_2$ and $M = (1)_2$.
- Representation:

1	01111001	100000000000000000000000
S	E	M

The number is stored in the computer as

10111100110000000000000000000000

Retrieving numbers stored in IEEE standard floating point format:

1. Find the value of S, E, and M.
2. If $S=0$, set the sign to positive, otherwise set the sign to negative.
3. Find the shifter ($E-127$).
4. De-normalize the mantissa.
5. Change the de-normalized number and find the absolute value (in decimal).
6. Add sign.

Example 4:

The bit pattern $(1\mathbf{10010100}00000000111000100001111)_2$ is stored in Excess_127 format, in memory. Show the retrieved value in decimal.

Solution

- a. The first bit represents S, the next eight bits, E and the remaining 23 bits, M.

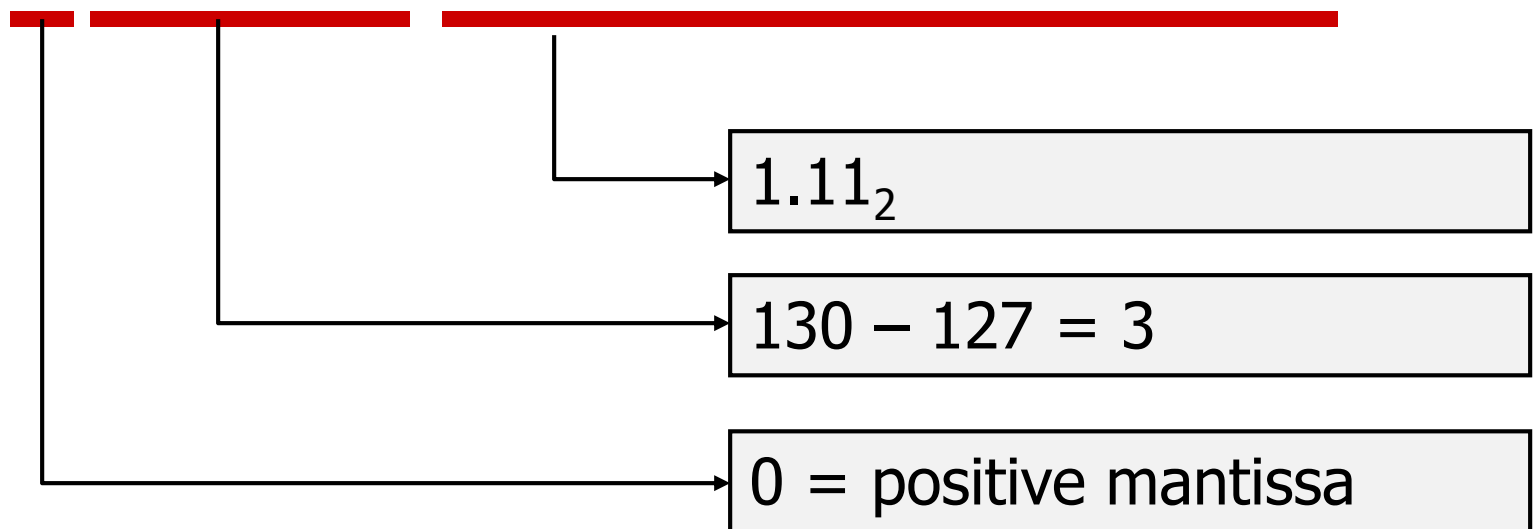
S	E	M
1	10010100	00000000111000100001111

- b. The sign is negative.
c. The shifter = $E - 127 = 148 - 127 = 21$.
d. This gives us $(1.00000000111000100001111)_2 \times 2^{21}$
e. The binary number is $(1000000001110001000011.11)_2$
f. The absolute value is 2,104,387.75
g. The number is $-2,104,387.75$

Example 5

- Single precision

0 10000010 110000000000000000000000



$$+1.11_2 \times 2^3 = 1110.0_2 = 14.0_{10}$$

Exercise

1. Represent $+0.8$ in the following floating-point representation:
 - 1-bit sign
 - 4-bit exponent
 - 6-bit normalised mantissa (significand).
2. Convert the value represented back to decimal.
3. Calculate the relative error of the representation.

Binary Codes

- **Computers also use binary numbers (integers) to represent non-numeric information, such as text or graphics (images).**
- **Binary representations of text, (letters, textual numbers, punctuation symbols, etc.) are called codes.**
- **In a binary code, the binary number is a symbol and does not represent an actual number.**
- **A code normally cannot be “operated on” in the usual fashion – mathematical, logical, etc. That is, one can not usually add up, for example, two binary codes. It would be like attempting to add text and graphics!**

Character representation- ASCII

- **ASCII** (American Standard Code for Information Interchange) - **Binary Codes**
- It is the scheme used to represent characters.
- Each character is represented using **7-bit** binary code.
- If **8-bits** are used, the first bit is always set to **0**

Numeric and Alphabetic Codes

■ **ASCII code**

- **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange
- an **alphanumeric code**
- each character represented by a 7-bit code
 - gives 128 possible characters
 - codes defined for upper and lower-case alphabetic characters, digits 0 – 9, punctuation marks and various non-printing control characters (such as carriage-return and backspace)

ASCII – examples

Symbol	decimal	Binary
7	55	00110111
8	56	00111000
9	57	00111001
:	58	00111010
;	59	00111011
<	60	00111100
=	61	00111101
>	62	00111110
?	63	00111111
@	64	01000000
A	65	01000001
B	66	01000010
C	67	01000011

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	\$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(72	48	H	104	68	h
9	09	Horizontal tab	41	29)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	
29	1D	Group separator	61	3D	=	93	5D]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

■ Representation schemes:

■ **Top layers - Character string to character sequence:**

Write each letter separately, enclosed in quotes. End string with '\0'.

Notation: enclose strings in double quotes

"Hello world"

```
'H' 'e' 'l' 'l' 'o' ' ' 'W' 'o' 'r' 'l' 'd' '\0'
```

■ **Bottom layer - Character to bit-string:**

Represent a character using the binary equivalent according to the ASCII table provided.

"SI"

'S' 'I' '\0'

010100110100100100000000

The colors are intended to help you read it; computers don't care that all the bits run together.

exercise

- Use the ASCII table to write the ASCII code for the following string of characters:
 - CIS110
 - $6=2*3$
 - Write your name in hexadecimal.

Unicode - representation

- **ASCII** code can represent only $128 = 2^7$ characters.
- It only represents the English Alphabet, numeric characters, few other characters plus some control characters.
- **Unicode** is designed to represent the worldwide printable and non printable characters.
- It uses 16 bits (or more) and can represent **65536** characters (or more).
- For compatibility, the first **128 Unicode** are the same as that of the **ASCII**.

Unicode cont'd..

- Let's consider how Ethiopia's character sets are represented
- The character set is called Ethiopic
- Range: 1200-1378 (in hexadecimal)
- Example character sets

Syllables

1200	ሀ	ETHIOPIC SYLLABLE HA
1201	ሁ	ETHIOPIC SYLLABLE HU
1202	ሂ	ETHIOPIC SYLLABLE HI
1203	ሃ	ETHIOPIC SYLLABLE HAA
1204	ሄ	ETHIOPIC SYLLABLE HEE
1205	ህ	ETHIOPIC SYLLABLE HE
1206	ሆ	ETHIOPIC SYLLABLE HO
1207	ሐ	ETHIOPIC SYLLABLE HOA
1208	ለ	ETHIOPIC SYLLABLE LA
1209	ሉ	ETHIOPIC SYLLABLE LU
120A	ሊ	ETHIOPIC SYLLABLE LI
120B	ላ	ETHIOPIC SYLLABLE LAA
120C	ሌ	ETHIOPIC SYLLABLE LEE
120D	ል	ETHIOPIC SYLLABLE LE
120E	ሎ	ETHIOPIC SYLLABLE LO
120F	ሏ	ETHIOPIC SYLLABLE LWA
1210	ሐ	ETHIOPIC SYLLABLE HHA
1211	ሑ	ETHIOPIC SYLLABLE HHU
1212	ሒ	ETHIOPIC SYLLABLE HHI
1213	ሓ	ETHIOPIC SYLLABLE HHAA
1214	ሔ	ETHIOPIC SYLLABLE HHEE
1215	ሕ	ETHIOPIC SYLLABLE HHE
1216	ሖ	ETHIOPIC SYLLABLE HHO
1217	ሐ	ETHIOPIC SYLLABLE HHWA
1218	መ	ETHIOPIC SYLLABLE MA
1219	ሙ	ETHIOPIC SYLLABLE MU

1242	ቂ	ETHIOPIC SYLLABLE QI
1243	ቃ	ETHIOPIC SYLLABLE QAA
1244	ቄ	ETHIOPIC SYLLABLE QEE
1245	ቅ	ETHIOPIC SYLLABLE QE
1246	ቆ	ETHIOPIC SYLLABLE QO
1247	ቇ	ETHIOPIC SYLLABLE QOA
1248	ቈ	ETHIOPIC SYLLABLE QWA
1249	␣	<reserved>
124A	ቊ	ETHIOPIC SYLLABLE QWI
124B	ቋ	ETHIOPIC SYLLABLE QWAA
124C	ቌ	ETHIOPIC SYLLABLE QWEE
124D	ቍ	ETHIOPIC SYLLABLE QWE
124E	␣	<reserved>
124F	␣	<reserved>
1250	ቐ	ETHIOPIC SYLLABLE QHA
1251	ቑ	ETHIOPIC SYLLABLE QHU
1252	ቒ	ETHIOPIC SYLLABLE QHI
1253	ቓ	ETHIOPIC SYLLABLE QHAA
1254	ቔ	ETHIOPIC SYLLABLE QHEE
1255	ቕ	ETHIOPIC SYLLABLE QHE
1256	ቆ	ETHIOPIC SYLLABLE QHO
1257	␣	<reserved>
1258	ቈ	ETHIOPIC SYLLABLE QHWA
1259	␣	<reserved>
125A	ቊ	ETHIOPIC SYLLABLE QHWI
125B	ቋ	ETHIOPIC SYLLABLE QHWAA
125C	ቌ	ETHIOPIC SYLLABLE QHWEE
125D	ቍ	ETHIOPIC SYLLABLE QHWE

exercise

- Use UNICODE character representation to write the following:

U U- U U U U U

Boolean Algebra & Digital Logic

- Boolean algebra is a mathematical system for the manipulation of variables that can have one of two values.
 - In formal logic, these values are “true” and “false.”
 - In digital systems, these values are “on” and “off,” 1 and 0, or “high” and “low.”
- Boolean expressions are created by performing operations on Boolean variables.
 - (Common Boolean operators include **AND**, **OR**, and **NOT**.)

one unary operator “not” (symbolized by an over bar), two binary operators “+” and “.”

Boolean Algebra

- A Boolean operation can be completely described using a truth table.
- The truth table for the Boolean operators AND and OR are shown at the right.
- The AND operation is also known as a Boolean product. The OR operation is the Boolean sum.

X AND Y

X	Y	XY
0	0	0
0	1	0
1	0	0
1	1	1

X OR Y

X	Y	X+Y
0	0	0
0	1	1
1	0	1
1	1	1

Boolean Algebra

- The truth table for the Boolean NOT operator is shown at the right.
- The NOT operation is most often designated by an overbar. It is sometimes indicated by a prime mark (') or an “elbow” (\neg).

NOT X

X	\overline{X}
0	1
1	0

Boolean Algebra

- A Boolean function has:
 - At least one Boolean variable,
 - At least one Boolean operator, and
 - At least one input from the set $\{0,1\}$.
- It produces an output that is also a member of the set $\{0,1\}$.

Most modern programming Languages
include the **Boolean** data type.

Logic Gates

- We have looked at Boolean functions in abstract terms.
- In this section, we see that Boolean functions are implemented in digital computer circuits called gates.
- A gate is an electronic device that produces a result based on two or more input values.
 - In reality, gates consist of one to six transistors, but digital designers think of them as a single unit.
 - Integrated circuits contain collections of gates suited to a particular purpose.

Logic Gates

- The three simplest gates are the AND, OR, and NOT gates.



X AND Y

X	Y	XY
0	0	0
0	1	0
1	0	0
1	1	1



X OR Y

X	Y	X+Y
0	0	0
0	1	1
1	0	1
1	1	1



NOT X

X	\bar{X}
0	1
1	0

- They correspond directly to their respective Boolean operations, as you can see by their truth tables.

Logic Gates

- Another very useful gate is the exclusive OR (XOR) gate.
- The output of the XOR operation is true only when the values of the inputs differ.

X XOR Y

X	Y	$X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0



Note the special symbol \oplus for the XOR operation.

Adders

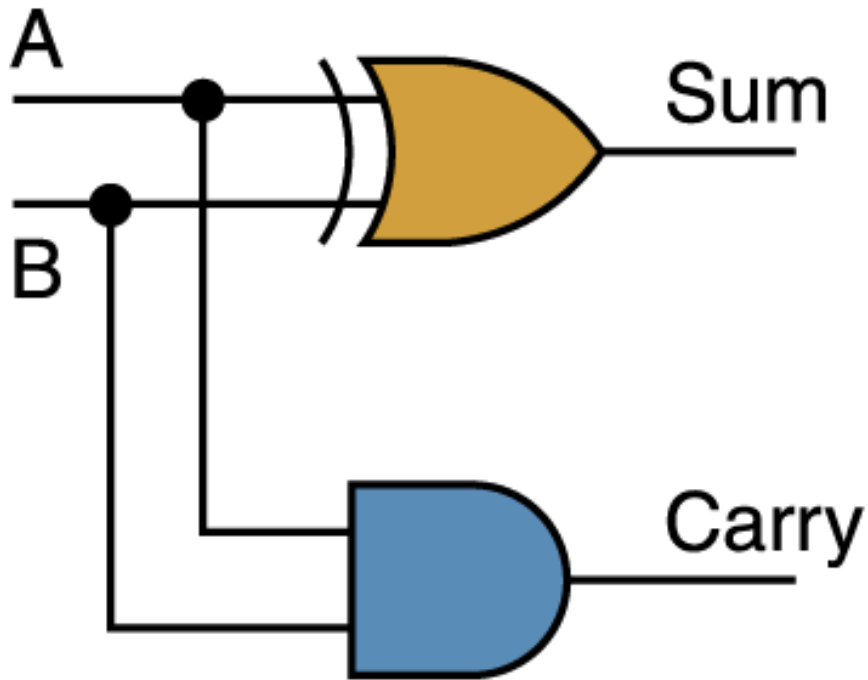
- At the digital logic level, addition is performed in binary
- Addition operations are carried out by special circuits called, appropriately, **adders**

Adders

- The result of adding two binary digits could produce a *carry value*
- Recall that $1 + 1 = 10$ in base two
- A circuit that computes the **sum** of two bits and produces the correct carry bit is called a **half adder**

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Adders

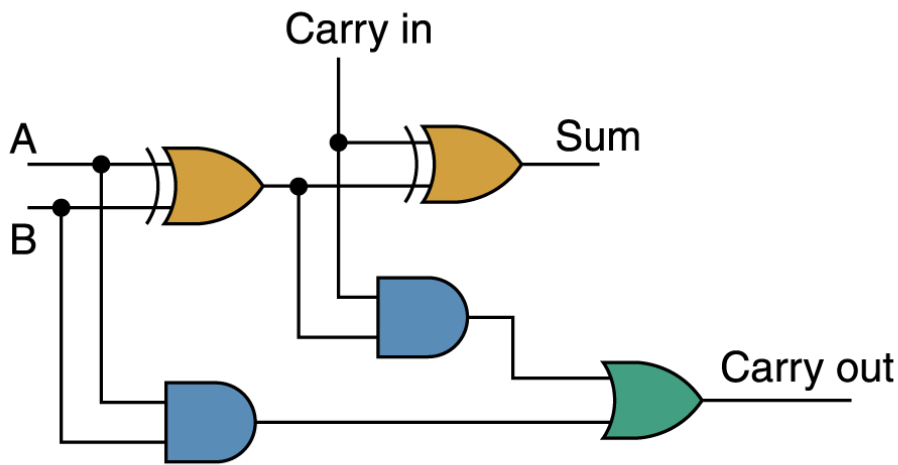


- Circuit diagram representing a **half adder**
- Two Boolean expressions:
$$\text{sum} = A \oplus B$$
$$\text{carry} = AB$$

Adders

- A circuit called a **full adder** takes the carry-in value into account

Logic Diagram

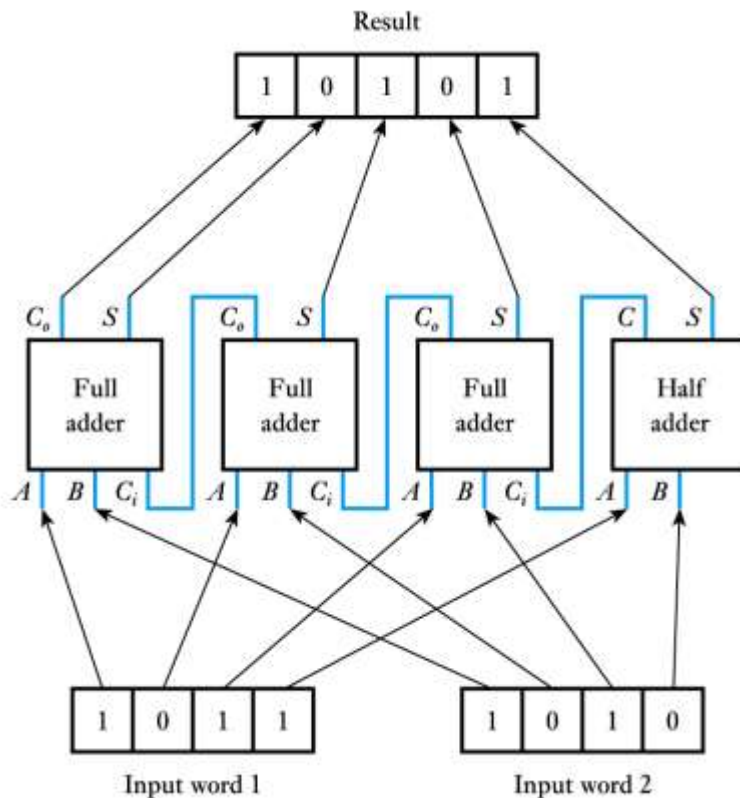


Truth Table

A	B	Carry-in	Sum	Carry-out
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

Full Adder

- More complex circuits can add digital words



- Similar circuits can be constructed to perform subtraction
- More complex arithmetic (such as multiplication and division) *can* be done by dedicated hardware but is more often performed using a microcomputer or complex logic device

Assignment:

- Construct a digital circuit that takes a 4-bit binary number as an input and outputs the 2's complement of the entered number.