

From Bits and Gates to C and Beyond

Bits, Data Types, and Operations

Chapter 2

How do we represent data in a computer?

At the lowest level, a computer is an electronic machine.

- Works by controlling the flow of **electrons**.

Easy to recognize two conditions:

1. Presence of a voltage – we'll call this state "1."
2. Absence of a voltage – we'll call this state "0."

We could base the state on the value of the voltage, but control and detection circuits are more complex.

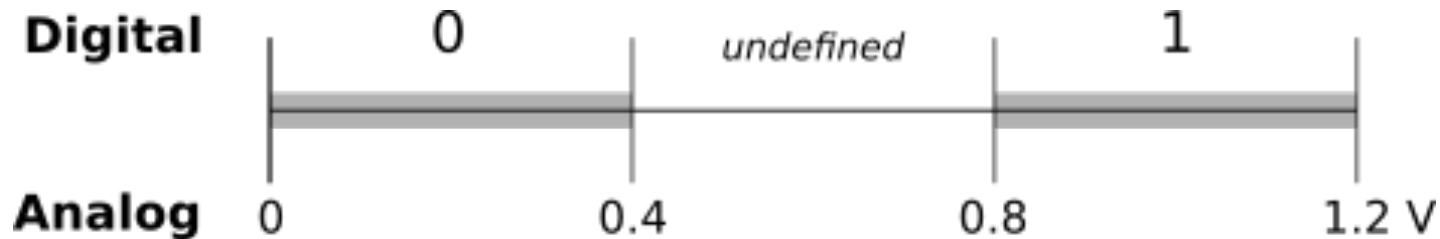
- Compare sticking a lightbulb (or your finger) in a socket versus using a voltmeter.

Computer is a Binary Digital system

Digital = finite number of values (compared to “analog” = infinite values)

Binary = only two values: 0 and 1

Unit of information = binary digit, or “bit”



Circuits (see Chapter 3) will pull voltage down towards zero, or will pull up towards the highest voltage.

Grey areas represent noise margin -- allowable deviation due to resistance, capacitance, interference from other circuits, temperature, etc. More reliable than analog.

[Access the text alternative for slide images.](#)

More than two values...

Each "wire" in a logic circuit represents one bit = 0 or 1.

Values with more than two states require multiple wires (bits).

With two bits, can represent four different values:

00, 01, 10, 11

With three bits, can represent eight different values:

000, 001, 010, 011, 100, 101, 110, 111

With **n** bits, can represent **2^n** different values.

What kinds of values do we want to represent?

Numbers - integer, fraction, real, complex, ...

Text - characters, strings...

Images - pixels, colors, shapes...

Sound

Video

Logical - true, false

Instructions

All data is represented as a collection of bits.

We encode a value by assigning a bit pattern to represent that value.

We perform operations (transformations) on bits, and we interpret the results according to how the data is encoded.

Data Type

In a computer system, we need a **representation** of data and **operations** that can be performed on the data by the machine instructions or the computer language.

This combination of *representation* + *operations* is known as a **data type**.

Type	Representation	Operations
Unsigned integers	binary	add, multiply, etc.
Signed integers	2's complement binary	add, multiply, etc.
Real numbers	IEEE floating-point	add, multiply, etc.
Text characters	ASCII	input, output, compare

Unsigned Integers ¹

Non-positional notation (unary)

- Could represent a number (“5”) with a string of ones (“11111”).
- Problems?

Weighted positional notation (binary)

- Like decimal numbers: “329.”
- “3” is worth 300, because of its position, while “9” is only worth 9.
- Since only 0 and 1 digits, use a binary (base-two) number system.

$$\begin{array}{ccc} & 329 & \\ / & | & \backslash \\ 10^2 & 10^1 & 10^0 \end{array}$$

$$3 \times 100 + 2 \times 10 + 9 \times 1 = 329$$

$$\begin{array}{ccc} \text{most} & & \text{least} \\ \text{significant} & \text{101} & \text{significant} \\ / & | & \backslash \\ 2^2 & 2^1 & 2^0 \end{array}$$

$$1 \times 4 + 0 \times 2 + 1 \times 1 = 5$$

[Access the text alternative for slide images.](#)

Unsigned Integers ₂

Encode a decimal integer using base-two binary number

Use **n** bits to represent values from 0 to $2^n - 1$

2^2	2^1	2^0	value
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

Unsigned Binary Arithmetic

Base-2 addition -- just like base-10

- Add from right to left, propagating carry.

$$\begin{array}{r} 10010 \\ + 1001 \\ \hline 11011 \end{array}$$
$$\begin{array}{r} 10010 \\ + 1011 \\ \hline 11101 \end{array}$$
$$\begin{array}{r} 1111 \\ + 1 \\ \hline 10000 \end{array}$$
$$\begin{array}{r} 10111 \\ + 111 \\ \hline \end{array}$$

Can also do subtraction, multiplication, etc., using base-2.

[Access the text alternative for slide images.](#)

Signed Integers

How to represent positive (+) and negative (–)?

With n bits, we have 2^n encodings.

- Use half of the patterns (the ones starting with zero) to represent positive numbers.
- Use the other half (the ones starting with one) to represent negative numbers.

Three different encoding schemes

- Signed-magnitude.
- 1's complement.
- 2's complement.

Signed-Magnitude

Leftmost bit represents the **sign** of the number.

0 = positive

1 = negative

Remaining bits represent the **magnitude** of the number using the binary notation used for unsigned integers.

Examples of 5-bit signed-magnitude integers:

00101 = +5 (sign = 0, magnitude = 0101)

10101 = -5 (sign = 1, magnitude = 0101)


01101 = +13 (sign = 0, magnitude = 1101)

10010 = -2 (sign = 1, magnitude = 0010)


1's complement

To get a negative number, start with a positive number (with zero as the leftmost bit) and flip all the bits -- from 0 to 1, from 1 to 0.

Examples -- 5-bit 1's complement integers:



00101 (5)
11010 (-5)



01001 (9)
10110 (-9)

[Access the text alternative for slide images.](#)

Disadvantages of Signed-Magnitude and 1's Complement

In both representations, two different representations of zero

- Signed-magnitude: $00000 = 0$ and $10000 = 0$.
- 1's complement: $00000 = 0$ and $11111 = 0$.

Operations are not simple.

- Think about how to add $+2$ and -3 .
- Actions are different for two positive integers, two negative integers, one positive and one negative.

A simpler scheme: 2's complement

2's Complement

To simplify circuits, we want all operations on integers to use binary arithmetic, regardless of whether the integers are positive or negative.

When we add $+X$ to $-X$ we want to get zero.

Therefore, we use "normal" unsigned binary to represent $+X$.

And we assign to $-X$ the pattern of bits that will make $X + (-X) = 0$.

$$\begin{array}{r} 00101 \quad (5) \\ + 11011 \quad (-5) \\ \hline 00000 \quad (0) \end{array}$$

$$\begin{array}{r} 01001 \quad (9) \\ + 10111 \quad (-9) \\ \hline 00000 \quad (0) \end{array}$$

NOTE: When we do this, we get a carry-out bit of 1, which is ignored.

We only have 5 bits, so the carry-out bit is ignored and we still have 5 bits.

[Access the text alternative for slide images.](#)

2's Complement Integers

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

With n bits, represent values from -2^{n-1} to $+2^{n-1} - 1$

NOTE: All positive number start with 0, all negative numbers start with 1.

Converting X to -X

1. Flip all the bits. (Same as 1's complement.)
2. Add +1 to the result.

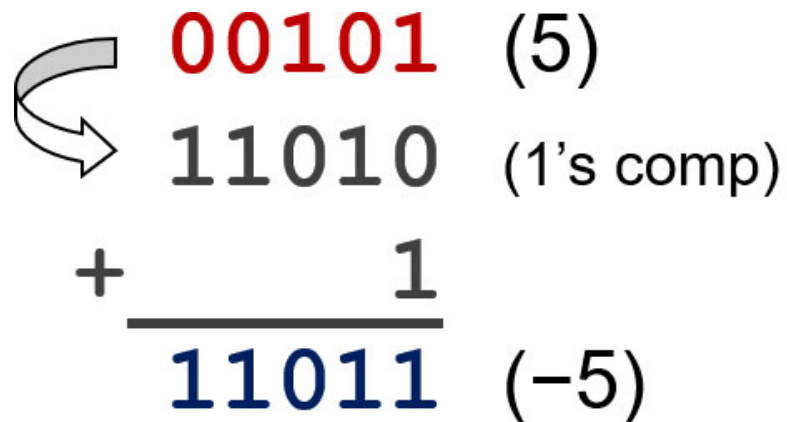


Diagram illustrating the conversion of 5 to -5 using 1's complement and adding 1:

$$\begin{array}{r} \text{00101} \quad (5) \\ \xrightarrow{\text{1's comp}} \text{11010} \quad (1\text{'s comp}) \\ + \quad \quad \quad 1 \\ \hline \text{11011} \quad (-5) \end{array}$$

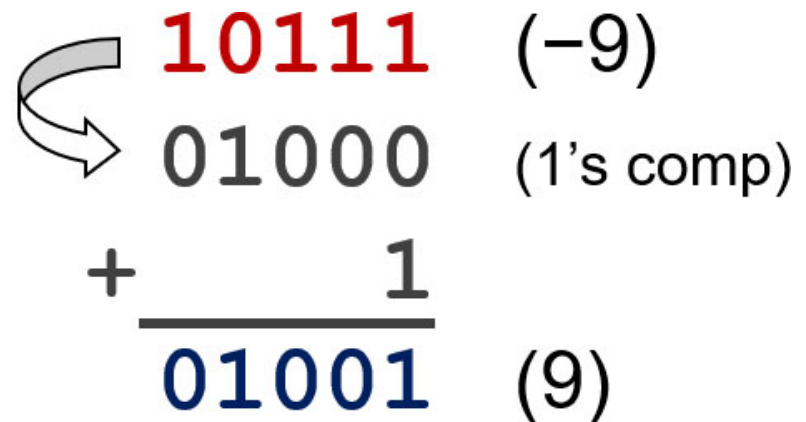


Diagram illustrating the conversion of -9 to 9 using 1's complement and adding 1:

$$\begin{array}{r} \text{10111} \quad (-9) \\ \xrightarrow{\text{1's comp}} \text{01000} \quad (1\text{'s comp}) \\ + \quad \quad \quad 1 \\ \hline \text{01001} \quad (9) \end{array}$$

A shortcut method:

Copy bits from right to left up to (and including) the first '1'.
Then flip remaining bits to the left.

Converting Binary (2's C) to Decimal

1. If leading bit is one, take two's complement to get a positive number.
2. Add powers of 2 that have "1" in the corresponding bit positions.
3. If original number was negative, add a minus sign.

$$\begin{aligned} X &= 01101000_{\text{two}} \\ &= 2^6 + 2^5 + 2^3 = 64 + 32 + 8 \\ &= 104_{\text{ten}} \end{aligned}$$

Examples use 8-bit 2's complement numbers.

<i>n</i>	<i>2ⁿ</i>
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

More Examples

$$X = 00100111_{\text{two}}$$

$$= 2^5 + 2^2 + 2^1 + 2^0 = 32 + 4 + 2 + 1$$

$$= 39_{\text{ten}}$$

$$X = 11100110_{\text{two}}$$

$$-X = 00011010$$

$$= 2^4 + 2^3 + 2^1 = 16 + 8 + 2$$

$$= 26_{\text{ten}}$$

$$X = -26_{\text{ten}}$$

n	2^n
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

Examples use 8-bit 2's complement numbers.

Converting Decimal to Binary (2's C) ₁

First Method: *Division*

1. Find magnitude of decimal number. (Always positive.)
2. Divide by two – remainder is least significant bit.
3. Keep dividing by two until answer is zero, writing remainders from right to left.
4. Append a zero as the MS bit; if original number was negative, flip bits and add +1.

$$X = 104_{\text{ten}}$$

$$104 / 2 = 52 \text{ } r0 \quad \textit{bit 0}$$

$$52 / 2 = 26 \text{ } r0 \quad \textit{bit 1}$$

$$26 / 2 = 13 \text{ } r0 \quad \textit{bit 2}$$

$$13 / 2 = 6 \text{ } r1 \quad \textit{bit 3}$$

$$6 / 2 = 3 \text{ } r0 \quad \textit{bit 4}$$

$$3 / 2 = 1 \text{ } r1 \quad \textit{bit 5}$$

$$1 / 2 = 0 \text{ } r1 \quad \textit{bit 6}$$

$$X = 01101000_{\text{two}}$$

Converting Decimal to Binary (2's C) ₂

Second Method: *Subtract Powers of Two*

1. Find magnitude of decimal number.
2. Subtract largest power of two less than or equal to number.
3. Put a one in the corresponding bit position.
4. Keep subtracting until result is zero.
5. Append a zero as MS bit; if original was negative, flip bits and add +1.

<i>n</i>	<i>2ⁿ</i>
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024

$$X = 104_{\text{ten}}$$

$$104 - 64 = 40$$

bit 6

$$40 - 32 = 8$$

bit 5

$$8 - 8 = 0$$

bit 3

$$X = 01101000_{\text{two}}$$

Operation: Addition

As discussed, 2's complement addition is just binary addition.

- Assume operands have the same number of bits.
- Ignore carry-out.

$$\begin{array}{rcl} & 01101000 & (104) \\ + & 11110000 & (-16) \\ \hline & 01011000 & (98) \end{array} \quad \begin{array}{rcl} & 11110110 & (-10) \\ + & 11110111 & (-9) \\ \hline & 11101101 & (-19) \end{array}$$

[Access the text alternative for slide images.](#)

Operation: Subtraction

You can, of course, do subtraction in base-2, but easier to negate the second operand and add.

Again, assume same number of bits, and ignore carry-out.

$\begin{array}{r} 01101000 \quad (104) \\ - 00010000 \quad (16) \\ \hline 01101000 \quad (104) \\ + 11110000 \quad (-16) \\ \hline 01011000 \quad (88) \end{array}$	$\begin{array}{r} 11110110 \quad (-10) \\ - 11110111 \quad (-9) \\ \hline 11110110 \quad (-10) \\ + 00001001 \quad (9) \\ \hline 11111111 \quad (-1) \end{array}$
---	---

[Access the text alternative for slide images.](#)

Operation: Sign Extension

To add/subtract, we need both numbers to have the same number of bits.
What if one number is smaller?

Padding on the left with zero does not work:

4-bit

0100 (+4)

1100 (-4)

8-bit

00000100 (still +4)

00001100 (12, not -4)

Instead, replicate the most significant bit (the sign bit):

4-bit

0100 (+4)

1100 (-4)

8-bit

00000100 (still +4)

11111100 (still -4)

[Access the text alternative for slide images.](#)

Overflow

If the numbers are too big, then we cannot represent the sum using the same number of bits.

For 2's complement, this can only happen if both numbers are positive or both numbers are negative.

$$\begin{array}{r} 01000 \quad (8) \\ + 01001 \quad (9) \\ \hline 10001 \quad (-15) \end{array}$$

$$\begin{array}{r} 11000 \quad (-8) \\ + 10111 \quad (-9) \\ \hline 01111 \quad (+15) \end{array}$$

How to test for overflow:

1. Signs of both operands are the same, AND.
2. Sign of sum is different.

Another test (easier to perform in hardware): Carry-in to most significant bit position is different than carry-out.

[Access the text alternative for slide images.](#)

Logical Operations: AND, OR, NOT

If we treat each bit as TRUE (1) or FALSE (0), then we define these logical operations, also known as Boolean operations, on a bit.

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

A	NOT A
0	1
1	0

View n -bit number as a collection of n logical values (*bit vector*)

- operation applied to each bit independently.

Examples of Logical Operations

AND

useful for clearing bits

- AND with zero = 0.
- AND with one = no change.

$$\begin{array}{r} \phantom{\text{AND}} \phantom{\underline{}} 11000101 \\ \text{AND } \underline{} 00001111 \\ \phantom{\text{AND}} \phantom{\underline{}} 00000101 \end{array}$$

OR

useful for setting bits

- OR with zero = no change.
- OR with one = 1.

$$\begin{array}{r} \phantom{\text{OR}} \phantom{\underline{}} 11000101 \\ \text{OR } \underline{} 00001111 \\ \phantom{\text{OR}} \phantom{\underline{}} 11001111 \end{array}$$

NOT

unary operation -- one argument

- flips every bit.

$$\begin{array}{r} \text{NOT } \underline{} 11000101 \\ \phantom{\text{NOT}} \phantom{\underline{}} 00111010 \end{array}$$

[Access the text alternative for slide images.](#)

Logical Operation: Exclusive-OR (XOR)

Another useful logical operation is the XOR.
True is one of the bits is 1, but not both.

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

XOR

useful for selectively flipping bits

- XOR with zero = no change.
- XOR with one = flip.

$$\begin{array}{r} 11000101 \\ \text{XOR } 00001111 \\ \hline 11001010 \end{array}$$

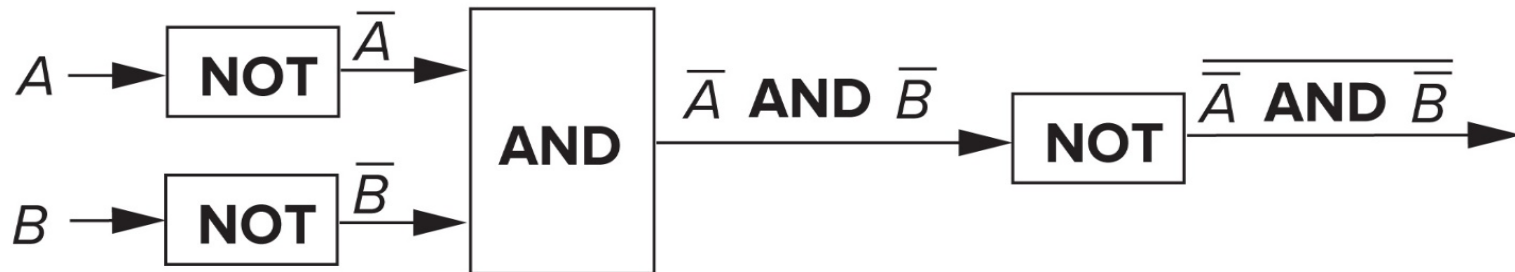
[Access the text alternative for slide images.](#)

DeMorgan's Laws ¹

There's an interesting relationship between AND and OR.

If we NOT two values (A and B), AND them, and then NOT the result, we get the same result as an OR operation. (In the figure below, an overbar denotes the NOT operation.)

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



Here's the truth table to convince you:

Copyright © McGraw Hill Education. Permission required or display

A	B	\bar{A}	\bar{B}	$\bar{A} \text{ AND } \bar{B}$	$\overline{\bar{A} \text{ AND } \bar{B}}$
0	0	1	1	1	0
0	1	1	0	0	1
1	0	0	1	0	1
1	1	0	0	0	1

[Access the text alternative for slide images.](#)

DeMorgan's Laws ₂

This means that any OR operation can be written as a combination of AND and NOT.

$$\begin{array}{r} \text{OR} \quad \begin{array}{r} 11000101 \\ 00001111 \\ \hline 11001111 \end{array} \end{array} \qquad \begin{array}{r} \text{AND} \quad \begin{array}{r} 00111010 \\ 11110000 \\ \hline 00110000 \end{array} \end{array}$$
$$\begin{array}{r} \text{NOT} \quad \begin{array}{r} 00110000 \\ \hline 11001111 \end{array} \end{array}$$

This is interesting, but is it useful? We'll come back to this in later chapters...

Also, convince yourself that a similar "trick" works to perform AND using only OR and NOT.

[Access the text alternative for slide images.](#)

Hexadecimal Notation: Binary Shorthand

To avoid writing long (error-prone) binary values, group four bits together to make a base-16 digit. Use A to F to represent values 10 to 15.

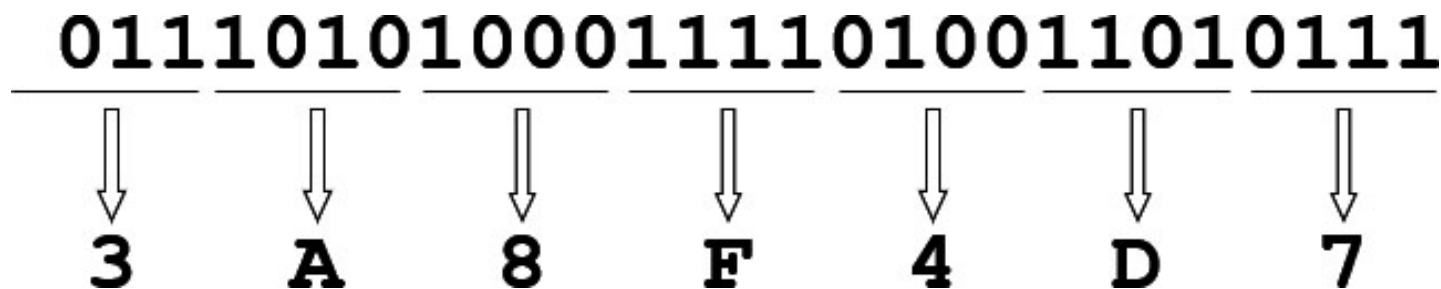
binary (base 2)	hexadecimal (base 16)	decimal (base 10)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7

binary (base 2)	hexadecimal (base 16)	decimal (base 10)
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	15
1111	F	15

Ex: Hex number **3D6E** easier to communicate than binary **0011110101101110**.

Converting from binary to hex

Starting from the right, group every four bits together into a hex digit. Sign-extend as needed.



This is not a new machine representation or data type, just a convenient way to write the number.

[Access the text alternative for slide images.](#)

Text: ASCII Characters

ASCII: Maps 128 characters to 7-bit code.

- both printable and non-printable (ESC, DEL, ...) characters

00	nul	10	dle	20	sp	30	0	40	@	50	P	60	`	70	p
01	soh	11	dc1	21	!	31	1	41	A	51	Q	61	a	71	q
02	stx	12	dc2	22	"	32	2	42	B	52	R	62	b	72	r
03	etx	13	dc3	23	#	33	3	43	C	53	S	63	c	73	s
04	eot	14	dc4	24	\$	34	4	44	D	54	T	64	d	74	t
05	enq	15	nak	25	%	35	5	45	E	55	U	65	e	75	u
06	ack	16	syn	26	&	36	6	46	F	56	V	66	f	76	v
07	bel	17	etb	27	'	37	7	47	G	57	W	67	g	77	w
08	bs	18	can	28	(38	8	48	H	58	X	68	h	78	x
09	ht	19	em	29)	39	9	49	I	59	Y	69	i	79	y
0a	nl	1a	sub	2a	*	3a	:	4a	J	5a	Z	6a	j	7a	z
0b	vt	1b	esc	2b	+	3b	;	4b	K	5b	[6b	k	7b	{
0c	np	1c	fs	2c	,	3c	<	4c	L	5c	\	6c	l	7c	
0d	cr	1d	gs	2d	-	3d	=	4d	M	5d]	6d	m	7d	}
0e	so	1e	rs	2e	.	3e	>	4e	N	5e	^	6e	n	7e	~
0f	si	1f	us	2f	/	3f	?	4f	O	5f	_	6f	o	7f	del

Interesting Properties of ASCII Code

What is relationship between a decimal digit ('0', '1', ...) and its ASCII code?

What is the difference between an upper-case letter ('A', 'B', ...) and its lower-case equivalent ('a', 'b', ...)?

Given two ASCII characters, how do we tell which comes first in alphabetical order?

Are 128 characters enough?
(<http://www.unicode.org/>)

No new operations -- integer arithmetic and logic.

Fractions: Fixed-Point Binary

We use a "binary point" to separate integer bits from fractional bits, just like we use the "decimal point" for decimal numbers.

Two's complement arithmetic still works the same.

$2^{-1} = 0.5$
 $2^{-2} = 0.25$
 $2^{-3} = 0.125$

$$\begin{array}{r} 00101000.101 \text{ (40.625)} \\ + 11111110.110 \text{ (-1.25)} \\ \hline 00100111.011 \text{ (39.375)} \end{array}$$

NOTE: In a computer, the binary point is implicit -- it is not explicitly represented. We just interpret the values appropriately.

[Access the text alternative for slide images.](#)

Converting Decimal Fraction to Binary (2's C)

1. Multiply fraction by 2.
2. Record one's digit of result, then subtract it from the decimal number.
3. Continue until you get 0.0000... or you have used the desired number of bits (in which case you have approximated the desired value).

$$\begin{array}{lll} X = 0.421_{\text{ten}} & 0.421 \times 2 = 0.842 & \text{bit } -1 = 0 \\ & 0.842 \times 2 = 1.684 & \text{bit } -2 = 1 \\ & 0.684 \times 2 = 1.368 & \text{bit } -3 = 1 \\ & 0.368 \times 2 = 0.736 & \text{bit } -4 = 0 \\ & & \text{until 0, or until no more bits...} \end{array}$$

$$X = 0.0110_{\text{two}} \text{ (if limited to 4 fractional bits)}$$

Very Large or Very Small Numbers: Floating-Point

Large values: 6.023×10^{23} -- requires 79 bits

Small values: 6.626×10^{-34} -- requires >110 bits

- Range requires lots of bits, but only four decimal digits of precision.

Use a different encoding for the equivalent of “scientific notation”: $F \times 2^E$

Need to represent F (*fraction*), E (*exponent*), and sign.

IEEE 754 Floating-Point Standard (32-bits):



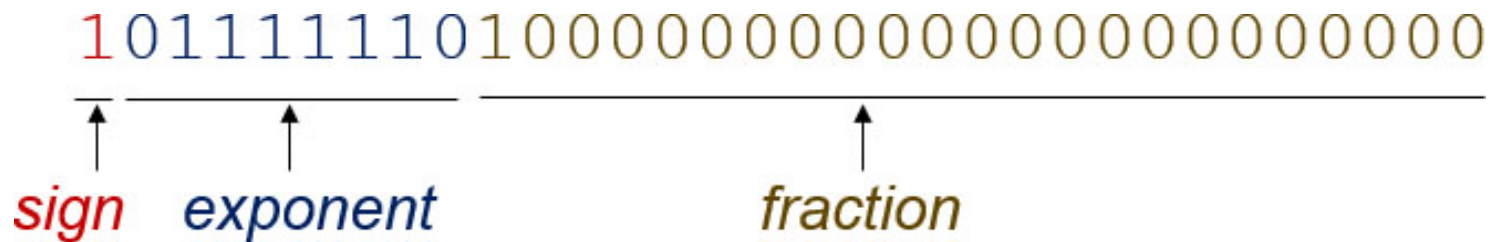
$$N = (-1)^S \times 1.\text{fraction} \times 2^{\text{exponent}-127}, \quad 1 \leq \text{exponent} \leq 254$$

$$N = (-1)^S \times 0.\text{fraction} \times 2^{-126}, \quad \text{exponent} = 0$$

[Access the text alternative for slide images.](#)

Floating-point Example

Single-precision IEEE floating point number:



- Sign is 1 – number is negative.
- Exponent field is 01111110 = 126 (decimal).
- Fraction is 0.1000000000000... = 0.5 (decimal).

$$\text{Value} = -1.5 \times 2^{(126-127)} = -1.5 \times 2^{-1} = -0.75.$$

[Access the text alternative for slide images.](#)

Converting from Decimal to Floating-Point Binary

How do we represent $-6 \frac{5}{8}$ as a floating-point binary number?

Ignoring sign, represent as a binary fraction: 0110.101

Next, normalize the number, by moving the binary point to the right of the most significant bit: $0110.101 = 1.10101 \times 2^2$

Exponent field = $127 + 2 = 129 = 10000001$

Fraction field = everything to the right of the binary point: 101010000....

Sign field is 1 (because the number is negative).

Answer:

11000000110101000000000000000000

Special Values

Infinity

- If exponent field is 11111111, the number represents **infinity**.
- Can be positive or negative (per sign bit).

Subnormal

- If exponent field is 00000000, the number is not normalized. This means that the implicit 1 is not present before the binary point. The exponent is -126 .
- $N = (-1)^s \times 0.\textit{fraction} \times 2^{-126}$.
- Extends the range into very small numbers.
- Example: 00000000000001000000000000000000.

$$0.00001_{\text{two}} \times 2^{-126} = 2^{-131}$$



Because learning changes everything.®

www.mheducation.com