

# Chapter 2

## Bits, Data Types, and Operations

## 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"

**Could base state on *value* of voltage,  
but control and detection circuits more complex.**

- compare turning on a light switch to measuring or regulating voltage

## Computer is a binary digital system.

Digital system:

- finite number of symbols

Binary (base two) system:

- has two states: 0 and 1



Basic unit of information is the *binary digit*, or *bit*.

Values with more than two states require multiple bits.

- A collection of **two** bits has **four** possible states:  
**00, 01, 10, 11**
- A collection of **three** bits has **eight** possible states:  
**000, 001, 010, 011, 100, 101, 110, 111**
- A collection of  $n$  bits has  $2^n$  possible states.

# What kinds of data do we need to represent?

- **Numbers** – signed, unsigned, integers, floating point, complex, rational, irrational, ...
- **Text** – characters, strings, ...
- **Images** – pixels, colors, shapes, ...
- **Sound**
- **Logical** – true, false
- **Instructions**
- ...

## Data type:

- *representation* and *operations* within the computer

We'll start with numbers...

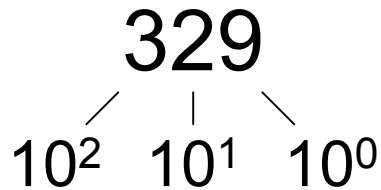
# Unsigned Integers

## Non-positional notation

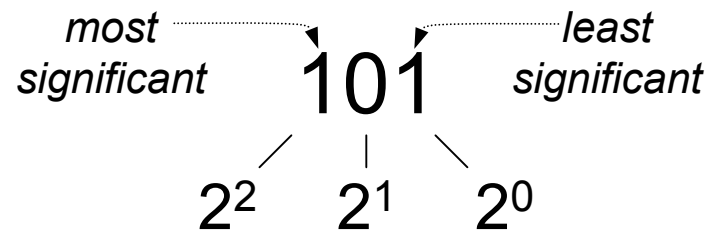
- could represent a number (“5”) with a string of ones (“11111”)
- problems?

## Weighted positional notation

- like decimal numbers: “329”
- “3” is worth 300, because of its position, while “9” is only worth 9



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



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

## Unsigned Integers (cont.)

An  $n$ -bit unsigned integer represents  $2^n$  values: from 0 to  $2^n-1$ .

$2^2$	$2^1$	$2^0$	
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

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

Subtraction, multiplication, division,...

## Signed Integers

With  $n$  bits, we have  $2^n$  distinct values.

- assign about half to positive integers (1 through  $2^{n-1}$ ) and about half to negative ( $-2^{n-1}$  through  $-1$ )
- that leaves two values: one for 0, and one extra

### Positive integers

- just like unsigned – zero in *most significant* (MS) bit  
 $00101 = 5$

### Negative integers

- sign-magnitude – set MS bit to show negative, other bits are the same as unsigned  
 $10101 = -5$
- one's complement – flip every bit to represent negative  
 $11010 = -5$
- in either case, MS bit indicates sign: 0=positive, 1=negative

## Two's Complement

### Problems with sign-magnitude and 1's complement

- two representations of zero (+0 and -0)
- arithmetic circuits are complex
  - How to add two sign-magnitude numbers?
    - e.g., try  $2 + (-3)$
  - How to add to one's complement numbers?
    - e.g., try  $4 + (-3)$

*Two's complement* representation developed to make circuits easy for arithmetic.

- for each positive number (X), assign value to its negative (-X), such that  $X + (-X) = 0$  with “normal” addition, ignoring carry out

	<b>00101</b>	(5)		<b>01001</b>	(9)
<b>+</b>	<b><u>11011</u></b>	(-5)	<b>+</b>		(-9)
	<b>00000</b>	(0)		<b>00000</b>	(0)



## Two's Complement Representation

If number is positive or zero,

- normal binary representation, zeroes in upper bit(s)

If number is negative,

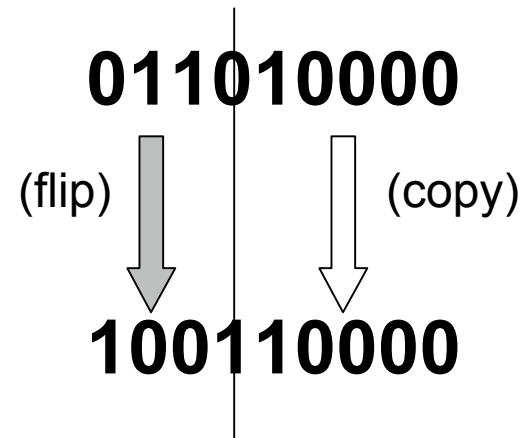
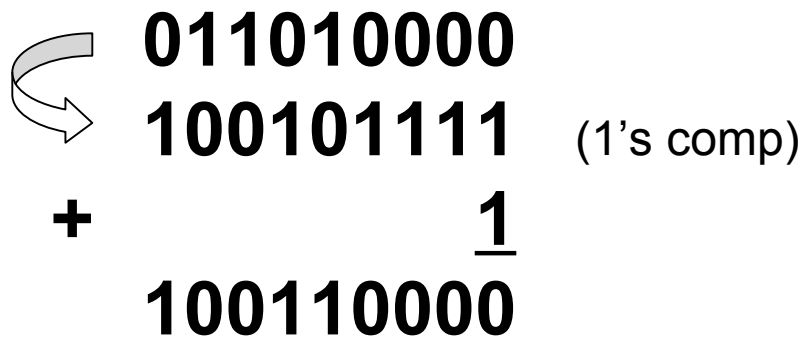
- start with positive number
- flip every bit (i.e., take the one's complement)
- then add one

	<b>00101</b>	(5)		<b>01001</b>	(9)
	<b>11010</b>	(1's comp)			(1's comp)
<b>+</b>	<b><u>1</u></b>		<b>+</b>	<b><u>1</u></b>	
	<b>11011</b>	(-5)			(-9)

## Two's Complement Shortcut

To take the two's complement of a number:

- copy bits from right to left until (and including) the first "1"
- flip remaining bits to the left



## Two's Complement Signed Integers

MS bit is sign bit – it has weight  $-2^{n-1}$ .

Range of an n-bit number:  $-2^{n-1}$  through  $2^{n-1} - 1$ .

- The most negative number ( $-2^{n-1}$ ) has no positive counterpart.

$-2^3$	$2^2$	$2^1$	$2^0$		$-2^3$	$2^2$	$2^1$	$2^0$	
0	0	0	0	0	1	0	0	0	-8
0	0	0	1	1	1	0	0	1	-7
0	0	1	0	2	1	0	1	0	-6
0	0	1	1	3	1	0	1	1	-5
0	1	0	0	4	1	1	0	0	-4
0	1	0	1	5	1	1	0	1	-3
0	1	1	0	6	1	1	1	0	-2
0	1	1	1	7	1	1	1	1	-1

## 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.

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

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

*Assuming 8-bit 2's complement numbers.*

## More Examples

$$\begin{aligned} X &= 00100111_{\text{two}} \\ &= 2^5 + 2^2 + 2^1 + 2^0 = 32 + 4 + 2 + 1 \\ &= 39_{\text{ten}} \end{aligned}$$

$$\begin{aligned} X &= 11100110_{\text{two}} \\ -X &= 00011010 \\ &= 2^4 + 2^3 + 2^1 = 16 + 8 + 2 \\ &= 26_{\text{ten}} \\ X &= -26_{\text{ten}} \end{aligned}$$

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

*Assuming 8-bit 2's complement numbers.*

## Converting Decimal to Binary (2's C)

### 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, take two's complement.

$X = 104_{\text{ten}}$	$104/2 = 52 \text{ r}0$	<i>bit 0</i>
	$52/2 = 26 \text{ r}0$	<i>bit 1</i>
	$26/2 = 13 \text{ r}0$	<i>bit 2</i>
	$13/2 = 6 \text{ r}1$	<i>bit 3</i>
	$6/2 = 3 \text{ r}0$	<i>bit 4</i>
	$3/2 = 1 \text{ r}1$	<i>bit 5</i>
$X = 01101000_{\text{two}}$	$1/2 = 0 \text{ r}1$	<i>bit 6</i>

## Converting Decimal to Binary (2's C)

### 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, take two's complement.

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

$X = 104_{\text{ten}}$	$104 - 64 = 40$	<i>bit 6</i>
	$40 - 32 = 8$	<i>bit 5</i>
	$8 - 8 = 0$	<i>bit 3</i>

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

## Operations: Arithmetic and Logical

Recall:

a data type includes *representation* and *operations*.

We now have a good representation for signed integers, so let's look at some arithmetic operations:

- **Addition**
- **Subtraction**
- **Sign Extension**

We'll also look at overflow conditions for addition.

Multiplication, division, etc., can be built from these basic operations.

Logical operations are also useful:

- **AND**
- **OR**
- **NOT**

## Addition

As we've discussed, 2's comp. addition is just binary addition.

- assume all integers have the same number of bits
- ignore carry out
- for now, assume that sum fits in n-bit 2's comp. representation

$$\begin{array}{r} \phantom{+} \quad \mathbf{01101000} \quad (104) \\ + \quad \mathbf{11110000} \quad (-16) \\ \hline \mathbf{01011000} \quad (98) \end{array} \quad \begin{array}{r} \phantom{+} \quad \mathbf{11110110} \quad (-10) \\ + \quad \phantom{0000} \quad (-9) \\ \hline \phantom{0000} \quad (-19) \end{array}$$

*Assuming 8-bit 2's complement numbers.*

## Subtraction

**Negate subtrahend (2nd no.) and add.**

- assume all integers have the same number of bits
- ignore carry out
- for now, assume that difference fits in n-bit 2's comp. representation

$$\begin{array}{r} \phantom{-} \mathbf{01101000} \ (104) \\ - \mathbf{00010000} \ (16) \\ \hline \mathbf{01101000} \ (104) \\ + \mathbf{11110000} \ (-16) \\ \hline \mathbf{01011000} \ (88) \end{array} \qquad \begin{array}{r} \phantom{-} \mathbf{11110110} \ (-10) \\ - \phantom{000} \phantom{0000} \ (-9) \\ \hline \mathbf{11110110} \ (-10) \\ + \phantom{000} \phantom{0000} \ (9) \\ \hline \phantom{000} \phantom{0000} \ (-1) \end{array}$$

*Assuming 8-bit 2's complement numbers.*

## Sign Extension

To add two numbers, we must represent them with the same number of bits.

If we just pad with zeroes on the left:

<u>4-bit</u>		<u>8-bit</u>	
<b>0100</b>	(4)	<b>00000100</b>	(still 4)
<b>1100</b>	(-4)	<b>00001100</b>	(12, not -4)

Instead, replicate the MS bit -- the sign bit:

<u>4-bit</u>		<u>8-bit</u>	
<b>0100</b>	(4)	<b>00000100</b>	(still 4)
<b>1100</b>	(-4)	<b>11111100</b>	(still -4)

## Overflow

If operands are too big, then sum cannot be represented as an  $n$ -bit 2's comp number.

$$\begin{array}{r} 01000 \quad (8) \\ + \underline{01001} \quad (9) \\ \hline 10001 \quad (-15) \end{array} \qquad \begin{array}{r} 11000 \quad (-8) \\ + \underline{10111} \quad (-9) \\ \hline 01111 \quad (+15) \end{array}$$

We have overflow if:

- signs of both operands are the same, and
- sign of sum is different.

Another test -- easy for hardware:

- carry into MS bit does not equal carry out

## Logical Operations

### Operations on logical TRUE or FALSE

- two states -- takes one bit to represent: TRUE=1, FALSE=0

A	B	A AND B	A	B	A OR B	A	NOT A
0	0	0	0	0	0	0	1
0	1	0	0	1	1	1	0
1	0	0	1	0	1		
1	1	1	1	1	1		

### View $n$ -bit number as a collection of $n$ logical values

- operation applied to each bit independently

## Examples of Logical Operations

### AND

- useful for clearing bits
  - AND with zero = 0
  - AND with one = no change

	<b>11000101</b>
AND	<u><b>00001111</b></u>
	<b>00000101</b>

### OR

- useful for setting bits
  - OR with zero = no change
  - OR with one = 1

	<b>11000101</b>
OR	<u><b>00001111</b></u>
	<b>11001111</b>

### NOT

- unary operation -- one argument
- flips every bit

	<b>11000101</b>
NOT	<u><b>11000101</b></u>
	<b>00111010</b>

## Hexadecimal Notation

It is often convenient to write binary (base-2) numbers as hexadecimal (base-16) numbers instead.

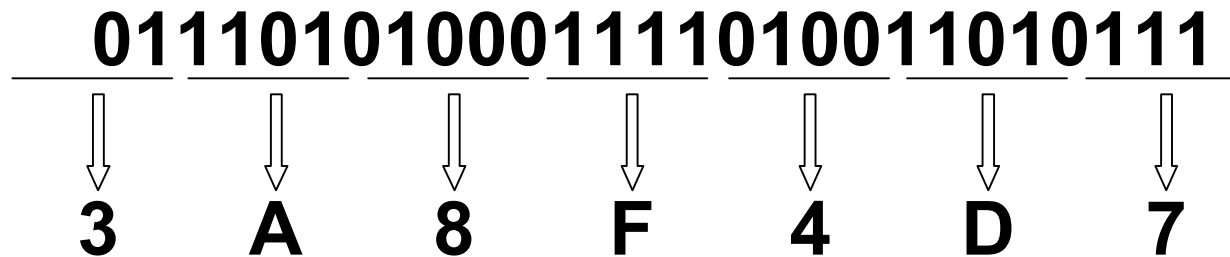
- fewer digits -- four bits per hex digit
- less error prone -- easy to corrupt long string of 1's and 0's

Binary	Hex	Decimal	Binary	Hex	Decimal
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	A	10
0011	3	3	1011	B	11
0100	4	4	1100	C	12
0101	5	5	1101	D	13
0110	6	6	1110	E	14
0111	7	7	1111	F	15

## Converting from Binary to Hexadecimal

Every four bits is a hex digit.

- start grouping from right-hand side



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

## Fractions: Fixed-Point

### How can we represent fractions?

- Use a “binary point” to separate positive from negative powers of two -- just like “decimal point.”
- 2’s comp addition and subtraction still work.
  - if binary points are aligned

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

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

*No new operations -- same as integer arithmetic.*

## Very Large and Very Small: Floating-Point

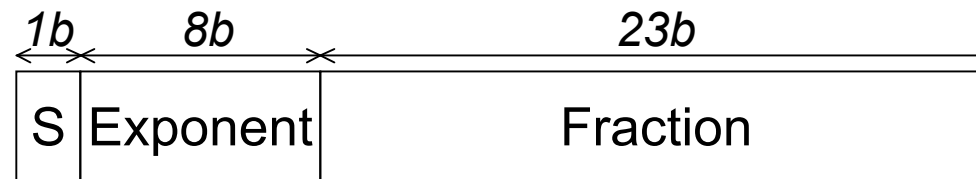
Large values:  $6.023 \times 10^{23}$  -- requires 79 bits

Small values:  $6.626 \times 10^{-34}$  -- requires >110 bits

Use equivalent of “scientific notation”:  $F \times 2^E$

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

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



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

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

## Floating Point Example

Single-precision IEEE floating point number:



- Sign is 1 – number is negative.
- Exponent field is 01111110 = 126 (decimal).
- Fraction is 0.100000000000... = 0.5 (decimal).

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

## Floating-Point Operations

**Will regular 2's complement arithmetic work for Floating Point numbers?**

**(Hint: In decimal, how do we compute  $3.07 \times 10^{12} + 9.11 \times 10^8$ ?)**

## 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.*

## Other Data Types

### Text strings

- **sequence of characters, terminated with NULL (0)**
- **typically, no hardware support**

### Image

- **array of pixels**
  - **monochrome: one bit (1/0 = black/white)**
  - **color: red, green, blue (RGB) components (e.g., 8 bits each)**
  - **other properties: transparency**
- **hardware support:**
  - **typically none, in general-purpose processors**
  - **MMX -- multiple 8-bit operations on 32-bit word**

### Sound

- **sequence of fixed-point numbers**

## LC-3 Data Types

**Some data types are supported directly by the instruction set architecture.**

**For LC-3, there is only one hardware-supported data type:**

- **16-bit 2's complement signed integer**
- **Operations: ADD, AND, NOT**

**Other data types are supported by interpreting 16-bit values as logical, text, fixed-point, etc., in the software that we write.**