Data Representation

Outline

- **Binary Numbers**
- Adding Binary Numbers
- Negative Integers
- Other Operations with Binary Numbers
- Floating Point Numbers
- Character Representation
- Image Representation
- Sound Representation

Bits and Bytes

- **bit**: Most basic unit of information in a computer. Two states:
  - 1: on, true
  - 0: off, false
- **byte**: A group of eight bits.
  - Often abbreviated using a capital B.
- **word**: A contiguous group of bytes.
  - The precise size of a word is machine-dependent.
  - Typically refers to the size of an address or integer.
  - The most common sizes are 32 bits (4 bytes) and 64 bits (8 bytes).

Number Representation

Initially, focus on non-negative integers of infinite length. Later in this unit, look at:

- fixed-length integers
- negative numbers
- floating point numbers (fractional component)

Base-10 Numbers

Our decimal system is a base-10 system: each digit represents a power of 10.

For instance, the decimal number 947 in powers of 10 can be expressed as:

$9*10^2 + 4*10^1 + 7*10^0$

$= 900 + 40 + 7$

$= 947$

Converting Binary into Decimal

Integers are stored in a computer in binary notation where each digit is a bit:

- Each bit represents a power of 2.
- Also called the base-2 system.
  - 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 = 1024; 2^11 = 2048

Converting Binary into Decimal Example

Example: Convert 101101_2 into base 10.

1*2^5 + 0*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0

= 32 + 0 + 8 + 4 + 0 + 1

= 45

Converting Decimal Numbers into Binary

How do you convert decimal numbers into binary? Two techniques:

- Subtraction Method
- Division/Remainder Method

The subtraction method is more intuitive than the division / remainder method but requires familiarity with the powers of 2.

Subtraction Method

Idea: Iteratively subtract the largest power of two.

Algorithm to convert decimal number n into binary number b:

1. Set $b = 0$.
2. Find x: largest power of 2 that does not exceed n.
3. Mark a 1 in the position represented by x in b.
4. $n = n - x$
5. If $n \neq 0$, repeat steps 2-4.

Subtraction Method Example

Example: Use the subtraction method to convert 90 into binary.

- X = ___ ; b = 0000000; N = 90
- X = 2^6 = 64; b = 1000000; N = 90 – 64 = 26
- X = 2^4 = 16; b = 1010000; N = 26 -16 = 10
- X = 2^3 = 8; b = 1011000; N = 10 -8 = 2
- X = 2^1 = 2; b = 1011010; N = 2 -2 = 0

- 90 = 1011010 in binary

Division / Remainder Method

Idea: Continuously divide by two and record the remainder.

Algorithm to convert decimal number n into binary number b:

1. Set $k = 0$, $b = 0$
2. Divide $n = n\ /\ 2$ storing the remainder (0 or 1) into r.
3. Bit 2^k of b is set to r.
4. $k = k + 1$
5. If $n \neq 0$, repeat steps 2 - 4.

Division / Remainder Method Example

Example: Use the division / remainder method to convert 177 into binary.

- N = 177/2 = 88; r = 1; b= 1
- N = 88/2 = 44; r = 0; b=01
- N = 44/2 = 22; r=0; b=001
- N = 22/2 = 11; r =0; b=0001
- N = 11/2 = 5; r=1; b=10001
- N = 5/2 = 2; r=1; b=110001
- N = 2/2 = 1; r =0; b = 0110001
- N = 1/2 = 0; r =1; b = 10110001
- 177 = 10110001 in binary

Hexadecimal Numbers

- Problem: Binary numbers can be long and difficult to read.
- Hexadecimal (base-16) numbers are often used to represent quantities in a computer.
  - Often preceded with '0x' such as 0x61A2F.
- Since $2^4 = 16$, it is easy to convert a binary number into hexadecimal:
  1. Divide the binary number into groups of four bits.
  2. Translate each four bit group into a hexadecimal digit.

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

Hexadecimal Number Example

Example: Convert the binary number into hexadecimal.

11010100011011

- 11010100011011 = 0x351B
- 11010100011011 = 32433 in Octal

Class Problem

Convert the number 489 into (a) binary and (b) hexadecimal.

- Binary: 111101001
- Hexadecimal: 0x1E9

Fixed Length Integers

- Integers in a computer have finite length.
- An unsigned (nonnegative) integer of n bits can represent values of 0 to $2^n - 1$.
- In C++, can find size of a type using `sizeof` operator. For instance, `sizeof(char)` $= 1$.

C++ Integer Sizes

Data type	*Size (bytes)*	*Unsigned range (0 to …)*
`char`	1	255
`short`	2	65,535
`int, long`	4	4,294,967,295
`long long`	8	~ 1.84 * 10^19

Outline

- Binary Numbers
- **Adding Binary Numbers**
- Negative Integers
- Other Operations with Binary Numbers
- Floating Point Numbers
- Character Representation
- Image Representation
- Sound Representation

Binary Math Facts

Works in the same way as base-10 addition. Math facts:

$0 + 0 = 0$

$0 + 1 = 1$

$1 + 0 = 1$

$1 + 1 = 10$

Binary Addition Example

Example: Find the sum of two bytes containing 139 and 46 using binary addition.

```
    10001011
+   00101110
------------
    10111001
------------
```

- 139 + 46 = 185

Binary Addition Example

Example: Now find the sum of two bytes containing 214 and 93 using binary addition.

```
   11010110
+  01011101
-----------
  100110011
-----------
```

- 214 + 93 = 307
- Outside the range of 1 byte, i.e., 255; overflow

Overflow

- **Arithmetic overflow** occurs when the result of an operation is too large to fit in the provided space.
- In many languages (including C++), overflow is undetected.
  - Responsibility of the programmer to check for and/or avoid overflow conditions.
  - This is especially problematic since many specifications are given using integers (infinite).
  - Potential security hazard if integer is used in pointer arithmetic or array references.
- Good rule of thumb: Restrict input as soon as enters the program.

Class Problem

Find the sums of these binary numbers. Assume a one-byte limit and indicate if overflow occurs.

```
    10100110          01011100
+   01101100      +   10001111
  ----------        ----------
   100010010          11101011
```

- Overflow in the first sum

Outline

- Binary Numbers
- Adding Binary Numbers
- **Negative Integers**
- Other Operations with Binary Numbers
- Floating Point Numbers
- Character Representation
- Image Representation
- Sound Representation

Signed Magnitude Numbers

Simple way of representing negative numbers: Reserve the left most bit to represent the sign.

- 0 is positive
- 1 is negative

Example: Represent –43 with one byte.

- 10101011

Signed Magnitude Numbers

Some issues with signed magnitude numbers:

- There are two representations of zero.
  - negative zero?
- Logic for dealing with sign is complicated.
  - Consider how you would add the numbers 34 + -78?

Two's Complement Numbers

Two's complement numbers arrange negative and positive numbers in an ordered number line.

-4	1111 1100
-3	1111 1101
-2	1111 1110
-1	1111 1111
0	0000 0000
1	0000 0001
2	0000 0010
3	0000 0011
4	0000 0100

Two's Complement Numbers

This creates new endpoints. For one byte the endpoints are:

- bottom (most negative): 1000 0000 (-128)
- top (most positive): 0111 1111 (127)

In general, if a number has *b* bits, the end points are:

- bottom (most negative): - 2^(b-1)
- top (most positive): 2^(b-1) - 1

Two's Complement Numbers

Why is there one more negative value than positive value?

- Zero consumes one of the positive value bit pattern

How do you determine if a value is positive or negative?

- Look at the left most bit, the sign bit
  - 1 => Negative
  - 0 => Zero or positive

Negating Two's Complement Numbers

To express a positive number – the representation is identical to unsigned.

- Remember that the range of positive numbers that can be represented is reduced.

To express a negative value – use this algorithm:

1. Start with the positive representation.
2. Flip the bits: 0→1, 1→0. (bitwise not)
3. Add 1.

Two's Complement Negation Example

Example: Express -43 in two's complement.

- Positive 43 = 0010 1011
- Flip bits: 1101 0100
- Add 1: 1101 0101

Negating Two's Complement Numbers

How do you convert a negative number to its positive representation? The same way!

1. Start with the negative representation.
2. Flip the bits.
3. Add 1.

Two's Complement Negation Example

Example: Express –(–43) in two's complement.

- Negative 43 = 1101 0101
- Flip the bits: 0010 1010
- Add 1: 0010 1011

Two's Complement Addition

Addition is carried out much the same way as unsigned numbers.

- No special work for negative numbers
- Only change is for overflow detection

Example: Add the numbers 75 and -39.

0100 1011	75
+ 1101 1001	+ -39
0010 0100	36

- No overflow

Two’s Complement Addition Overflow

Rule for detecting overflow when adding two's complement numbers: When the “carry in” and the “carry out” of the sign bit are different, overflow has occurred.

Example: Add the numbers 107 + 46.

```
  0110 1011        107
+ 0010 1110      +  46
-----------      -----
  1001 1001        153
-----------      -----
```

- Carry in of sign bit = 1
- Carry out of sign bit = 0
- Therefore an overflow!

Two’s Complement Overflow Cases

<u>Case 1: Adding a positive and a negative number.</u>

- The sign bits must be different (1 and 0)
- The carry out bit will always be the same as carry in
- Hence overflow can never occur

Two's Complement Overflow Cases

<u>Case 2: Adding two positive numbers.</u>

- The sign bits are both zero
- The carry out will be zero
- If the carry in is one
  - The result of 7-bit unsigned addition doesn't fit in 7 bits
  - Overflow has occured

Two's Complement Overflow Cases

Case 3: Adding two negative numbers.

- The sign bits are both one
- Carry out will always be one
- If carry in is zero
  - Overflow has occurred

Class Problem

Find the sums of these two's complement binary numbers. Assume a one-byte limit and indicate if overflow occurs.

```
  1001 1001        1011 0100        0010 0111
+ 0110 0111      + 1101 1010      + 0101 1001
 10000 0000       11000 1110       01000 0000
```

- Overflow in the third case:
  - Carry in = 1
  - Carry out = 0

Outline

- Binary Numbers
- Adding Binary Numbers
- Negative Integers
- **Other Operations with Binary Numbers**
- Floating Point Numbers
- Character Representation
- Image Representation
- Sound Representation

Numbering Bits

Bits are commonly numbered from right to left starting with 0:

bit 7 → 0100 1011 ← bit 0

- The rightmost bit (bit 0) is called the ***least significant bit***.
- The leftmost bit (bit n–1) is called the ***most significant bit***.
- Bits to the right are ***lower*** than bits to the left.

Sign Extension

- For two's complement numbers, to convert shorter (fewer bits) to longer numbers:
  1. Starting with bit 0, copy the shorter number bit by bit.
  2. When you are out of bits, replicate the sign bit (most significant bit) to the remaining bit positions.
- The process of replicating the sign is called ***sign extension***.
- For unsigned numbers, simply place a zero in all new bit positions.
  - This is called ***zero extension***.

Sign Extension Example

Example: Convert a byte containing -39 to a 16 bit number.

-39 in 8 bits: 1101 1001

-39 in 16 bits: 1111 1111 1101 1001

Other Arithmetic Operations

- ***Subtraction:*** Simply negate the subtrahend and add.
- ***Multiplication:*** Convert to positive numbers and determine sign at end.
  - Use grade-school (long) multiplication.
  - For multiplying two n-bit numbers, need $2n$ bits to represent the product.
- ***Division:***
  - Need to be careful about dividing by zero.
- All operations (including addition) have faster algorithms that are beyond the scope of the course.

Outline

- Binary Numbers
- Adding Binary Numbers
- Negative Integers
- Other Operations with Binary Numbers
- **Floating Point Numbers**
- Character Representation
- Image Representation
- Sound Representation

Floating Point Numbers

To represent non-integral numbers, computers use **floating point numbers**.

- In base-10 speak, such numbers have a *decimal point*.
- Since computers represent everything in binary, we could call it a *binary point*.
- However, the more generic term *floating point* is more commonly used.

Scientific Notation

Computers use a form of scientific notation for floating-point representation. Numbers written in scientific notation have three components:

- -3.83 * 10^4
  - "-" is the sign
  - 3.83 is the significand
    - 1<= significand < 10 (unless the number is exactly zero)
  - 4 is the exponent
- -1.00101 * 2^6
  - "-" is the sign
  - 1.00101 is the significand
    - 1 <= significand < 2 (unless the number is exactly zero)
  - 6 is the exponent

IEEE Floating Point Representation

Sign	Exponent	Fraction (similar to significand)
1 bit	8 bits	23 bits

- The one-bit *sign* field is the sign of the stored value.
    - 0 is positive, 1 is negative
- The size of the *exponent* determines the range of values that can be represented.
- The size of the *fraction* determines the precision of the representation.

Converting Decimal to Floating Point

To convert a decimal number into floating point requires three steps:

1. Convert the decimal number into a binary number.
2. Express the floating point number in scientific notation.
3. Fill in the various fields of the floating point number appropriately.

To illustrate this process, we will convert the number 10.625 into a IEEE floating point number.

Converting Decimal to Floating Point

<u>Step 1. Convert the decimal number into a binary number.</u>

Just like a base-10 number with a decimal point, the bits past the floating point represent negative powers of two:

$$\ldots b_3 b_2 b_1 b_0 . b_{-1} b_{-2} b_{-3} \ldots =$$
$$\ldots + b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0 2^0 + b_{-1} 2^{-1} + b_{-2} 2^{-2} + b_{-3} 2^{-3} + \ldots$$

where $2^{-1} = ½ = 0.5$, $2^{-2} = ¼ = 0.25$, $2^{-3} = ⅛ = 0.125$, …

Note: Some numbers such as 0.1 and 1/3 cannot be exactly represented in binary notation regardless of how many bits past the floating point are specified.

Converting Decimal to Floating Point

Example: Convert 10.625 into a binary number.

- $10.625 = 8 + 2 + 0.5 + 0.125 = 1010.101$

Converting Fraction to Floating Point

1. Begin with the decimal fraction F
2. Multiply the fraction F by two; F = F*2
3. Whole number part "W" of the multiplication result in step 2 is the next binary digit to the right of the point
4. Update F by discarding the whole number part; F = F – W
5. If F > 0, repeat steps 2, 3, 4

Converting Decimal to Floating Point

<u>Step 2. Express the floating point number in scientific notation.</u>

Recall in base-10 scientific notation, the number to the left of the decimal point must be 1-9 (unless the number is zero). Examples:

857.63 = 8.5763 * 10^2

0.00007634 = 7.634 * 19^(-5)

In binary, the number to the left of the floating point must be a 1 (unless the number is zero). Examples:

110111.01 = 1.1011101 * 2^5

0.011101 = 1.1101 * 2^(-2)

Converting Decimal to Floating Point

Example: Express 10.625 as a binary number in scientific notation.

- 1010.101 = 1.010101 * 2^3

Converting Decimal to Floating Point

Step 3. Fill in the various fields of the floating point number appropriately.

- Sign bit: 0 positive, 1 negative
- Exponent: Holds the exponent using a biased notation (more below).
- Fraction: Holds the fractional part of the significand in scientific notation.
  - The '1' before the floating point is implied and not stored.
  - Cannot represent zero (more later).

Converting Decimal to Floating Point

Exponent is stored using an unusual biased representation. Think of it as a combination of two's complement and unsigned numbers:

- Two's complement: number line including both negative and positive numbers
- Unsigned: lowest number is all zeroes, highest number is all ones.

Both of these properties are true in the biased representation.

Biased Representation

For an 8 bit exponent:

-127	0000 0000	reserved for special numbers
-126	0000 0001	
…	…	
-2	0111 1101	
-1	0111 1110	
0	0111 1111	
1	1000 0000	
2	1000 0001	
…	…	
127	1111 1110	
128	1111 1111	reserved for special numbers

Excess 127 Bias

- For 8 bits, this form is called ***excess 127 bias*** because the numbers are 127 apart from the two's complement equivalent.
- To convert a two's complement number to excess 127 bias: Add +127 in its two's complement form (0111 1111) and ignore overflow.
- To convert a number from excess 127 bias to two's complement: Add -127 in its two's complement form (1000 0001) and ignore overflow.

Converting Decimal to Floating Point

Example: Convert 10.625 as a IEEE floating point number in both binary and hexadecimal.

- 10.625 = 1010.101 (binary)
  - 1.010101 * 2^3
- Sign bit = 0
- Fraction = 010101…0 (23 bits)
- Exponent = 3 = 011 (binary) = 1000 0010 (Excess 127)
- Therefore the floating point number is
  - 0100 0001 0010 1010 0000 0000 0000 0000
  - 0x412A0000

Class Problem

Convert the IEEE floating point number 0xC2AC8000 into decimal.

- 1100 0010 1010 1100 1000 0000 0000 0000
  - Sign: 1
  - Exponent: 10000101
  - Fraction: 010 1100 1000 0000 0000 0000
- To convert exponent from excess 127 to two's complement add -127 (1000 0001) to it
  - Exponent = 6
- Fraction has an implied "1" and binary point, therefore:
  - The number is 1.01011001 * 2^6 = 1010110.01
  - 1010110.01 = 86.25
- As sign bit is 1, the decimal number is -86.25

Special Floating Point Numbers

Some bit patterns are reserved for special numbers:

zero	infinity	NaN (not a number)
• sign bit can be anything • exponent is all zeroes • fraction is all zeroes	• sign bit: 1(-∞), 0(+∞) • exponent is all ones • fraction is all zeroes	• sign bit can be anything • exponent is all ones • fraction is anything except all zeroes

Floating Point Approximations and Errors

- Since the number of bits is finite, not every real number can be represented.
- Many values cannot be represented exactly. This introduces error or imprecision in each floating point value and calculation.
- By using a greater number of bits in the fraction, the magnitude of the error is reduced but errors can never totally be eliminated.

Floating Point Terminology

- The ***range*** of a numeric format is the difference between the largest and smallest values that is can express. 32-bit IEEE FP range: 1.2×10^{-38} to 3.4×10^{38}
- ***Accuracy*** refers to how closely a numeric representation approximates a true value.
- The ***precision*** of a number indicates how much information we have about a value; the number of significant digits.
- ***Overflow*** occurs when there is no room to store the high-order bits resulting from a calculation.
- ***Underflow*** occurs when a value is too small to store, possibly resulting in division by zero.

Floating Point Error

- It is the programmer's job to reduce error or be aware of the magnitude of error during calculations.
- When testing floating point values for equality to zero or some other number, you need to figure out how close the numbers can be to be considered equal.

Replace: `if (a == b) …`

With:

```
fp_error = a - b;
if (abs(fp_error) < epsilon) …
```

Floating Point Error

Must be aware that errors can compound through repetitive arithmetic operations:

- The order of operations can affect the error.
- Associative, commutative or distributive laws may no longer apply.

Best practice: use operands similar in magnitude.

Floating Point Error Example

```
int main()
{
  int i;
  float a, b;

  a = 0.0;
  a = a + 10000000;
  for (i = 0; i < 20000000; i++) {
    a = a + 0.5;
  }
  cout << "a = " << a << endl;

  b = 0.0;
  for (i = 0; i < 20000000; i++) {
    b = b + 0.5;
  }
  b = b + 10000000;
  cout << "b = " << b << endl;

  return 0;
}
```

Example: What does this program print out?

Floating Point Addition

1. Check for any special values: zero, infinity, NaN.
2. Shift value with smaller exponent right to match larger exponent.
3. Add two values (or subtract if signs are different).
4. Normalize the value (in the form 1.bbb) and update exponent.
5. Check for zero and overflow.

Floating Point Multiplication

1. Check for any special values: zero, infinity, NaN.
2. Fill in sign based on sign of the two values; ignore sign for remaining steps.
3. Multiply fractions.
4. Multiply exponents (add them together).
5. Normalize the product (in the form 1.bbb) and update exponent.
6. Check for overflow and underflow.

Outline

- Binary Numbers
- Adding Binary Numbers
- Negative Integers
- Other Operations with Binary Numbers
- Floating Point Numbers
- **Character Representation**
- Image Representation
- Sound Representation

Character Representation

Characters (letters, digits, symbols) are represented using a code where each bit pattern represents a unique character.

Two common formats (virtually all machines use either or both of these formats):

- ASCII (American Standard Code for Information Interchange) is a 7-bit code.
- Unicode is a 16-bit code.
  - First 128 bit patterns are same as ASCII.
  - Includes letters and characters from non-English alphabets.
  - Includes more symbols (including math).

ASCII Character Set

Hex	Char	Hex	Char	Hex	Char	Hex	Char	Hex	Char	Hex	Char
20	(Space)	30	0	40	@	50	P	60	‘	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	’	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

- *Source: Andrew Tanenbaum*

ASCII Character Set

Hex	Name	Meaning	Hex	Name	Meaning
0	NUL	Null	10	DLE	Data Link Escape
1	SOH	Start Of Heading	11	DC1	Device Control 1
2	STX	Start Of Text	12	DC2	Device Control 2
3	ETX	End Of Text	13	DC3	Device Control 3
4	EOT	End Of Transmission	14	DC4	Device Control 4
5	ENQ	Enquiry	15	NAK	Negative AcKnowledgement
6	ACK	ACKnowledgement	16	SYN	SYNchronous idle
7	BEL	BELl	17	ETB	End of Transmission Block
8	BS	BackSpace	18	CAN	CANcel
9	HT	Horizontal Tab	19	EM	End of Medium
A	LF	Line Feed	1A	SUB	SUBstitute
B	VT	Vertical Tab	1B	ESC	ESCape
C	FF	Form Feed	1C	FS	File Separator
D	CR	Carriage Return	1D	GS	Group Separator
E	SO	Shift Out	1E	RS	Record Separator
F	SI	Shift In	1F	US	Unit Separator

- *Source: Andrew Tanenbaum*

Outline

- Binary Numbers
- Adding Binary Numbers
- Negative Integers
- Other Operations with Binary Numbers
- Floating Point Numbers
- Character Representation
- **Image Representation**
- Sound Representation

Image Representation

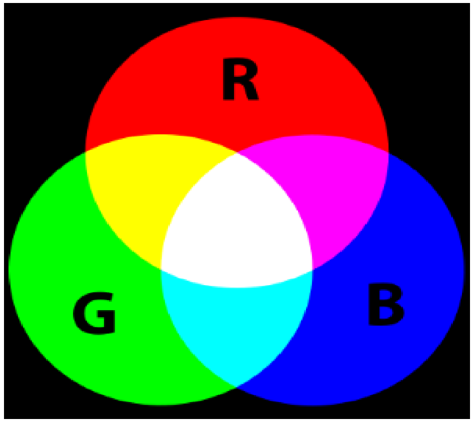
- Images can be thought of as a 2-dimensional array of pixels.
- Each pixel is a small dot within the image that has been assigned a color.
  - The pixels are small enough that the human eye is unable to detect the boundaries between the different pixels.
- A color is commonly represented using an RGB-value. RGB is a color model that produces colors by adding Red, Green, and Blue components.
  - Commonly, one byte (8 bits) is used for each of the three colors for a total of 24 bits for each pixel.
  - This model works best for computer monitors and televisions.

RGB Color Model

In the RGB color model, colors go from (0,0,0) to (255,255,255):

- (0,0,0) is black
- (255, 0, 0) is red
- (0, 255, 0) is green
- (0, 0, 255) is blue
- (0, 255, 255) is cyan
- (255, 0, 255) is magenta
- (255,255,0) is yellow
- (255,255,255) is

If all three color components are the same value → shade of gray



Source:Wikipedia

Image Representation

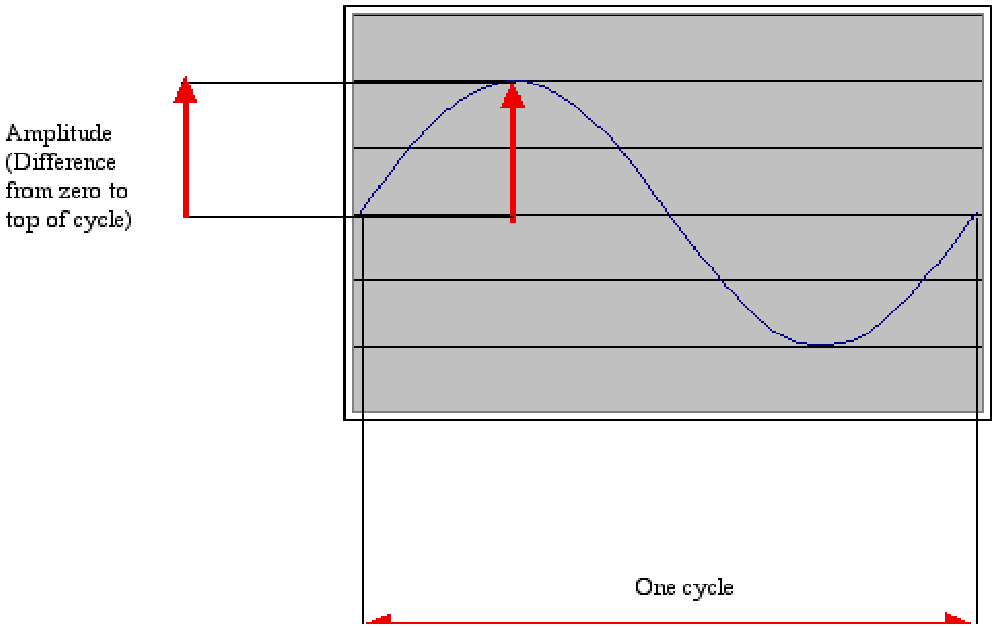
- With 24 bits – there are 16,777,216 (2^{24}) possible colors. However, computer monitors are unable to display 16 million colors.
- Color printers use a different color model – CYMK (Cyan, Yellow, Magenta, blacK).
- Images are typically stored in a compressed format (such as JPEG). Compression algorithms take advantage of the fact that pixels near one another are often close to the same color.

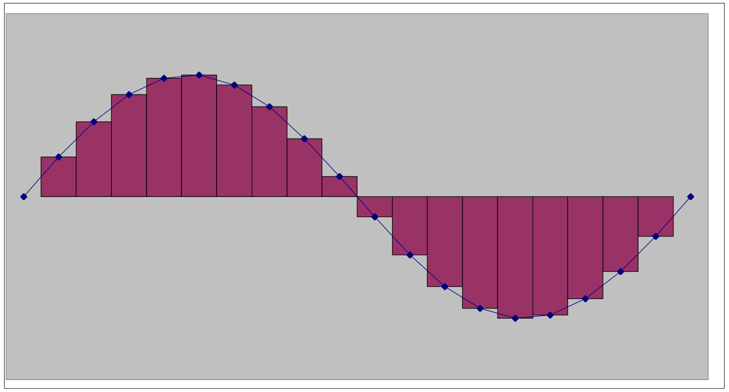
Outline

- Binary Numbers
- Adding Binary Numbers
- Negative Integers
- Other Operations with Binary Numbers
- Floating Point Numbers
- Character Representation
- Image Representation
- **Sound Representation**

Sound Representation

Sounds, in the physical world, are waves of air pressure. To digitize the sound wave curve, we need to sample the wave periodically, measuring the instantaneous amplitude.





Source: Mark Guzdial, Georgia Tech

Sampling

- The ***Nyquist–Shannon sampling theorem*** states that if the highest frequency of a sound is N Hertz, a sampling rate of at least 2N can perfectly reconstruct the original sound.
- The human ear can detect sounds up to 22,000 Hz approximately.
- CD quality sound is captured at 44,100 samples per second.
- Each sample is commonly encoded using 16 bits (a two's complement number).
- Like images, audio formats use compression.

Sound Representation Example

Example: How much memory is needed to store a 30-second audio file (uncompressed)?

- 44,100 samples per second
- Each sample has 16 bits
- Hence for 30 second recording we need:
  - 44100 * 16 * 30 bits
  - ~2.52 MB

Thank You!