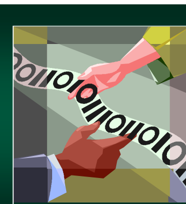
Part 1

Data



Binary Numbers

Bit of This and a Bit of That

What is a Number?

- We use the Hindu-Arabic Number System
  - positional grouping system
  - each position represents a power of 10
- Binary numbers
  - based on the same system
  - use powers of **2** rather than 10

Base 10 Number

The number 1783 is ...

10^4	10^3	10^2	10^1	10^0
10000	1000	100	10	1
0	**1**	**7**	**8**	**3**

1000 + 700 + 80 + 3 = 1783

Binary Number Example

The number 1010 1001 is ...

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
1	**0**	**1**	**0**	**1**	**0**	**0**	**1**

128 + 32 + 8 + 1 = 169

Binary Number Example

The number 1101 1011 is ...

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
1	**1**	**0**	**1**	**1**	**0**	**1**	**1**

128 + 64 + 16 + 8 + 2 + 1 = 219

Hexadecimal Numbers

- Writing out long binary numbers is cumbersome and error prone
- As a result, computer scientists often write computer numbers in hexadecimal
- Hexadecimal is base-16
  - We only have 0...9 to represent digits
  - So, hexadecimal uses **A...F** to represent **10...15**

Hexadecimal Numbers

Hex	Decimal	Binary
0	0	`0000`
1	1	`0001`
2	2	`0010`
3	3	`0011`
4	4	`0100`
5	5	`0101`
6	6	`0110`
7	7	`0111`

Hex	Decimal	Binary
8	8	`1000`
9	9	`1001`
A	10	`1010`
B	11	`1011`
C	12	`1100`
D	13	`1101`
E	14	`1110`
F	15	`1111`

Hex Example

The number **`A2C`** is ...

16^3	16^2	16^1	16^0
4096	256	16	1
0	**A**	**2**	**C**

$$(10\times256) + (2\times16) + (12\times1) = 2604$$

Converting Binary to Hex = Easy

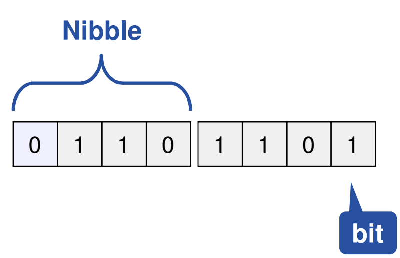
- Since $16 = 2^4$, a single hex character can represent a total of 4 bits
- Byte can be represented with only 2 hex digits!
- When looking at raw data, editors, called Hex Editors, display data as groups of 2 hex digits

5				**C**			
0	1	0	1	1	1	0	0

Bits and Bytes

- Everything in a *modern* computer is stored using combination of ones and zeros
- *Bit* is one binary digit
  - either 1 or 0
  - shorthand for a bit is b
- *Byte* is a group of 8 bits
  - e.g. `0010 0100`
  - shorthand for a byte is B

The Byte



Hex & Binary Notation

It gets confusing quick, so let's prepare

Hex & Binary Notation

- Hexadecimal and binary notations use the same digits we use for decimal
- As a result, some numbers look like valid hex, decimal and binary numbers

Hex & Binary Notation

- For example is **`101`** …
  - binary value 5 *?*
  - decimal value 101 *?*
  - hexadecimal value 257 *?*
- This, obviously, can become problematic

Subscript Notation

- Commonly, textbooks use a subscript to denote the base
- Examples
  - 101_{16} – hexadecimal, and equal to 257
  - 101_{2} – binary, and equal to 5
  - 101 – decimal
- However, this is not possible to do in common text editors

Postfix Character Notation

- One notation is to use postfix character for binary and hexadecimal numbers
- If no character is present, decimal is assumed
- "b" identifies the number as binary
- "h" identifies them as hexadecimal

Postfix Character

- Examples
  - 101**h** – hexadecimal, and equal to 257
  - 101**b** – binary, and equal to 5
  - 101 – just decimal
- Remember to use a lower case "b"
  - "B" is the hex digit for 11
  - someone could read 101B has hex

Prefix Notation

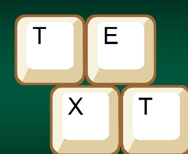
- There are also prefix notations that are commonly used.
- Using prefix characters "b" and "h"…
  - **h**101 – hexadecimal
  - **b**101 – binary
  - 101 – just decimal

C-Style Prefix Notation

- The C Programming Language's notation is often used
- C is hugely popular and multiple languages are based on its syntax – e.g. Java, C#

C-Style Prefix Notations

- C's notation
  - the prefix "0x" denotes hexadecimal
  - but it lacks a binary notation
  - so, "0b" typically denotes binary
- Examples:
  - 0x101 is hexadecimal
  - 0b101 is binary
  - 101 is decimal

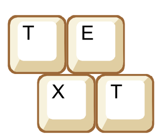


Text in Programming Languages

Press Any Key to Continue

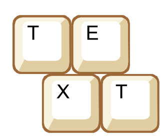
Characters

- Computer often store and transmit textual data
- Examples:
  - punctuation
  - numerals 0 – 9
  - letter
- Each of these symbols is called a *character* and are the basis for written communication



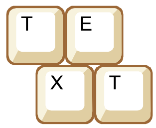
Characters

- Processors rarely know what a "character" is, and instead store each as an integer
- In this case, each character is given a unique value
- The letter "A", for instance, could have the value of 1, "B" is 2, etc…



Characters

- Characters and their matching values are a *character set*
- There have been many characters sets developed over time



Character Sets

- ASCII
  - 7 bits – 128 characters
  - uses a full byte, one bit is not used
  - created in the 1967
- EBCDIC
  - Alternative system used by old IBM systems
  - Not used much anymore

ASCII Chart

Control characters

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	sp	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Useful Control Characters

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	**NUL**	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	**HT**	**LF**	VT	FF	**CR**	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	sp	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

ASCII Codes

- Each character has a unique value
- The following is how "OMG" is stored in ASCII

	Binary	Hex	Decimal
O	0100 1111	4F	79
M	0100 1101	4D	77
G	0100 0111	47	71

ASCII Codes

- ASCII is laid out very logically
- Alphabetic characters (uppercase and lowercase) are 32 "code points" apart

	Binary	Hex
A	01000001	41
a	01100001	61

ASCII Codes

- 32 = 2^5
- Uppercase and lowercase letters are just 1 bit different
- Converting between the two is easy

	Binary	Hex
A	01000001	41
a	01100001	61

ASCII: Number Characters

- ASCII code for 0 is 30h
- The characters 0 to 9 can be easily converted to their binary values
- Notice that the binary value is stored in the lower nibble

0	0011 0000
1	0011 0001
2	0011 0010
3	0011 0011
4	0011 0100
5	0011 0101
6	0011 0110
7	0011 0111
8	0011 1000
9	0011 1001

ASCII: Number Characters

- Character → Binary
  - clear the upper nibble
  - Binary-And 0000 1111
- Binary → Character
  - set the upper nibble to 0011
  - Binary-Or 0011 0000

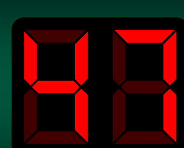
0	0011 0000
1	0011 0001
2	0011 0010
3	0011 0011
4	0011 0100
5	0011 0101
6	0011 0110
7	0011 0111
8	0011 1000
9	0011 1001

Unicode Character Set

- ASCII is only good for the United States
  - Other languages need additional characters
  - Multiple competing character sets were created
- Unicode was created to support every spoken language
- Developed in Mountain View, California

Unicode Character Set

- Originally used 16 bits
  - that's over 65,000 characters!
  - includes every character used in the World
- Expanded to 21 bits
  - 2 million characters!
  - now supports every character ever created
- Unicode can be stored in different formats

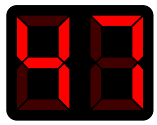


Numbers in Programs

The primitive types are pretty primitive

Primitive Data Types

- Most popular program languages hide the true nature of the computer from you
- However, most of the language's primitive data types are the same types recognized by the processor



Integer Data Types

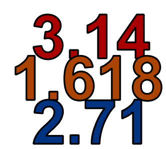
- Integer data types are stored in simple binary numbers
- The number of bytes used varies: 1, 2, 4, etc.…
- Languages often have a unique name for each – short, int, long, etc…

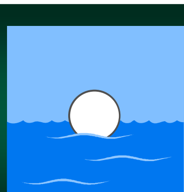
Floating-Point Data Type

- Floating-point numbers are usually stored using the IEEE 754 standard
- Languages often have unique names for them such as float, double, real

Floating-Point Data Type

- This is not always the case
  - some languages implement their own structures
  - e.g. COBOL
- Why?
  - some processors do not have floating-point instructions
  - or the language needs more precision and control



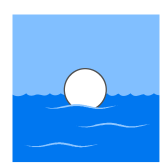


Floating Point Numbers

Real numbers are *real* complex

Floating Point Numbers

- Often, programs need to perform mathematics on *real* numbers
- *Floating point numbers* are used to represent quantities that cannot be represented by integers



Floating Point Numbers

- Why?
  - regular binary numbers can only store whole positive and negative values
  - many numbers outside the range representable within the system's bit width (too large/small)

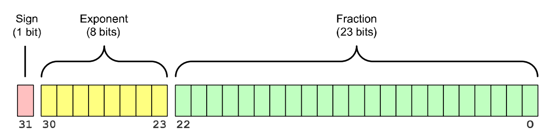
IEEE 754

- Practically modern computers use the *IEEE 754 Standard* to store floating-point numbers
- Represent by a mantissa and an exponent
  - similar to scientific notation
  - the value of a number is: $mantissa \times 2^{exponent}$
  - uses signed magnitude

IEEE 754

- Comes in three forms:
  - single-precision: 32-bit
  - double-precision: 64-bit
  - quad-precision: 128-bit
- Also supports special values:
  - negative and positive *infinity*
  - and "not a number" for errors (e.g. 1/0)

IEEE 754 Single Precision (32 bit)



Interpretation: Invalid Numbers

NaN → **1 / 0**

Naan →