

CMPT 130

Introduction to Computing Science and Programming I

What this course is about...

- Introduces basic ideas of computing science
- Describes the working principles of a computing machine (computer)
 - Information representation, Storage, and Processing
- Introduces basic concepts of communicating with a computer
 - Programming (**C++ Programming Language**)
- Introduces computational formulation of methods to solve given problems: Pseudocodes and Algorithms

What is Computing Science?

- The study of computing machines (computers) and computational systems
- A scientific and practical approach to computation and its applications
- A science of problem solving using computers
 - Abstraction, modeling and algorithms
 - Acquisition, representation, storage, processing, communication and access of information

How Does a Computer work?

- A computer is a **DIGITAL ELECTRONIC** machine!
 - As such, it understands only ‘certain’ **NUMBERS!!!**
- Really?
 - How are we then able to work with alphabets, symbols, images, videos, sound and much more things on our computers?
- Answer!
 - Everything in a computer is represented by unique combinations of numbers. (Example the letter ‘a’ is nothing but 97. We will come to this later)

Decimal Number System

- Let us begin with the Decimal Number System.
 - Numbers we (humans) use on our daily life
 - Ten unique digits (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9)
 - Every other number is represented by combining these unique digits
 - Why exactly 10 digits? Because we have 10 fingers and before the days of calculators we used our fingers for counting
 - But how many fingers does a computer have?

Decimal Number System (continued)

- How do we represent numbers outside the range of the basic digits in the decimal number system?
 - Write a number as a string of symbols.
 - Each symbol is one of the ten digits (0 – 9)
 - Interpret the number by multiplying each digit by a power of 10 associated with that digit's position
- Example:
 - The number $2578 = 2 \cdot 10^3 + 5 \cdot 10^2 + 7 \cdot 10^1 + 8 \cdot 10^0$

Decimal Number System (cont.)

- Decimal number system is called base-10 system
- The right most digit is least significant digit (exponent 0) and the left most digit is the most significant digit
- The exponent associated with each digit increments by 1 as we go from right most digit to the left
- Leading zeros do not have effect: $00762 = 762$
- Negative numbers are represented by prefixing the negative sign (-) to a number

What about a computer?

- Now, consider computers: made up of transistors.
- Transistors can only be **OFF** or **ON**.
- Therefore a computer can understand only two states (OFF or ON states of its millions of transistors)
- These two states are represented by **0** and **1**
- Thus a computer can understand only two numbers 0 and 1. (Computer has only two fingers!)
- Therefore we use binary number system (base-2) to represent any information in computers

Unsigned Binary Number System

- Consider the binary number **1101**. What number in decimal system does it represent?
 - It represents: $1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8+4+0+1 = 13$
- In the binary number system
 - Each symbol is called a BIT
 - In decimal the base is 10, in binary it is 2
 - As in decimal, interpreting a binary string entails multiplying each bit by a power of 2 associated with that bit's position

Examples

- Express the following binary numbers in decimal:

1111, 10001, 11100101

- Solution

$$1111 = 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 8+4+2+1 = 15$$

$$10001 = 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 16+0+0+0+1 = 17$$

$$11100101 = 1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 128 + 64 + 32 + 4 + 1 = 229$$

- Remember:** powers of 2 start at 0 for the right most bit and increment by 1 as we go left

Expressing Unsigned Decimal Numbers in Binary

- So, we saw that **11100101** = 229. But how do we go from 229 to express it as binary?

Procedure to convert decimal to binary

Step 1: Divide the decimal by 2 and record the remainder.


Step 2: Continue dividing the newest quotient obtained by 2; recording the remainders until a quotient of 0 is obtained.

Step 3: The binary representation is then the remainders recorded; starting from the last remainder going all the way to the first remainder in the order they were recorded.

Expressing Unsigned Decimal Numbers in Binary (cont.)

- Example: convert 229 to binary
- Solution:

<u>Operation</u>	<u>Quotient</u>	<u>Remainder</u>
229 / 2	114	1 ... Least significant bit
114 / 2	57	0
57 / 2	28	1
28 / 2	14	0
14 / 2	7	0
7 / 2	3	1
3 / 2	1	1
1 / 2	0	1 ... Most significant bit

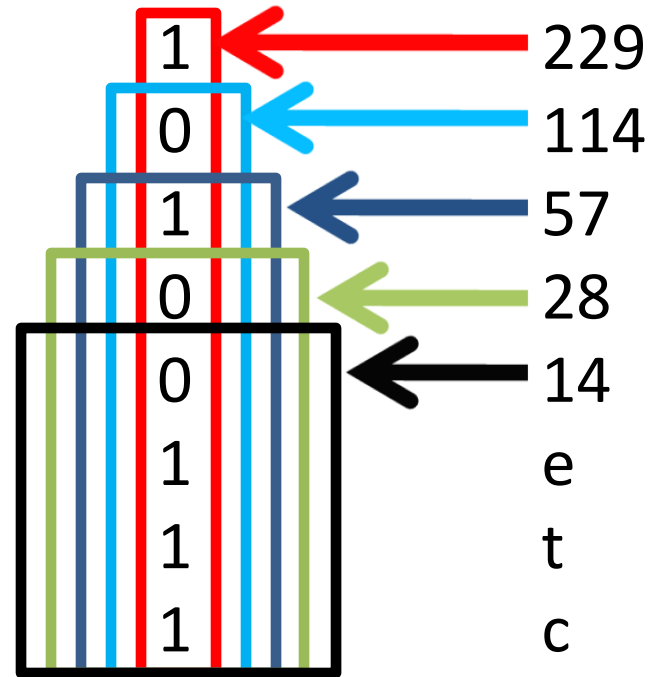


- The answer is therefore **11100101** as expected
- **Question:** What are the binary representations of 57, 7, 114?

Expressing Unsigned Decimal Numbers in Binary (cont.)

- Observation

<u>Operation</u>	<u>Quotients</u>	<u>Remainders</u>
229 / 2	114	1 ← 229
114 / 2	57	0 ← 114
57 / 2	28	1 ← 57
28 / 2	14	0 ← 28
14 / 2	7	0 ← 14
7 / 2	3	1
3 / 2	1	1
1 / 2	0	1



Bit Patterns

- Grouping bits together yields bit patterns common in computers.
 - Group of 1 bit Bit
 - Group of 4 bits Nibble
 - Group of 8 bits Byte
 - Group of 16 bits Word
 - Group of 32 bits Double Word
- Example: the decimal 13 is **1101** in Nibble, **00001101** in Byte and **00000000000000001101** in Word

Representing Unsigned Decimal Numbers in Bit Patterns

- Consider the Nibble (4-bit pattern). What are the decimal numbers that can be represented with it?
 - It is easy to see that the numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14 and 15 can be represented with a Nibble because their binary representations are: **0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, and 1111** respectively.
 - The decimal 16 can't be represented by a Nibble because its binary representation **10000** needs five bits which the Nibble can not handle
 - Thus we see that in an unsigned binary representation, a Nibble can represent 16 different unsigned decimal numbers namely 0, 1, 2, ..., 15

Representing Unsigned Decimal Numbers in Bit Patterns (cont.)

- Generally, a bit pattern of n bits can represent the decimal numbers **0** through **2^n-1**
- Thus
 - A Bit can represent the decimals 0 to $2^1-1 = 0$ to 1
 - A Nibble can represent the decimals 0 to $2^4-1 = 0$ to 15
 - A Byte can represent the decimals 0 to $2^8-1 = 0$ to 255
- **Remark:** The number of decimal numbers that can be represented by a bit, nibble and byte is respectively 2, 16 and 256. (Don't forget to count the decimal 0!)
- Generally, n bit pattern can represent **2^n** decimal numbers. These decimals are **0, 1, 2, ..., (2^n-1)**

Negative Decimal Numbers

- So far, we have seen only positive numbers and zero. Their binary representations are called **unsigned binary numbers**
- But what about negative numbers (signed numbers)?
 - In Decimal Number System, prefixing a number with the symbol minus (-) sign makes it negative
 - In a computer, there is no notion of minus sign. The only basic units are 0 and 1 bits
- One way of representing negative numbers is with **Sign and Magnitude Representation** where the left most bit is reserved for sign

Sign and Magnitude Representation

- Given a bit pattern (nibble, byte, word,...), reserve the left most bit for sign and the remaining bits for the actual decimal number (magnitude)

0 → positive and 1 → negative

- Therefore considering the nibble, we have

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

Example

- Give the sign and magnitude representation of the signed decimal number -45
 - a) As a seven bit pattern
 - b) As a byte
 - c) As a word
- Solution: Since the unsigned binary representation of the magnitude, which is 45, is given by 101101; we have
 - a) In seven bit, -45 as a sign and magnitude representation is 1101101
 - b) As a byte, it will 10101101
 - c) As a word, it will be 1000000000101101

Sign and Magnitude Representation

- How many decimal numbers can be represented by a given bit pattern using sign and magnitude representation?
- As shown above, a nibble can represent the signed integers -7 up to +7. Therefore it can represent 15 signed integers. Count them!
- Generally, an **n-bit** pattern can represent the signed integers **$-(2^{n-1}-1)$ up to $(2^{n-1}-1)$**

Sign and Magnitude Representation (cont.)

- Drawbacks

- It represents the decimal zero as two different bit patterns
- In order to do arithmetic on such binary bit patterns, we first need to identify which number is negative and which one is positive for correct arithmetic. This is extra work for a computer
- After doing an arithmetic, one would need to assert the sign bit is correctly set depending on the result. Again, extra work for the computer

Problem

- How can we represent signed integers in binary such that
 - The system represents zero as a unique bit pattern
 - Arithmetic on the system follows a standard arithmetic procedure with no additional task to handle signs
 - The result of any arithmetic will have the correct sign
- Solution
 - We use **Two's Complement Representation!**

Two's Complement Representation

- An elegant way to avoid the problems of signs and ambiguity in representing zero.
- All modern computers represent numbers with two's complement notation
- To find the two's complement of a given binary number, first flip all the digits and then add 1 to the result
- But how do we perform addition in binary number system?

Two's Complement Representation (cont.)

- Binary addition (single digit)

$\begin{array}{r} 0 \\ + 0 \\ \hline 0 \end{array}$	$\begin{array}{r} 1 \\ + 0 \\ \hline 1 \end{array}$	$\begin{array}{r} 0 \\ + 1 \\ \hline 1 \end{array}$	$\begin{array}{r} 1 \\ + 1 \\ \hline 10 \end{array}$
---	---	---	--

- Binary addition (bit patterns)

$\begin{array}{r} 1101 \\ + 1011 \\ \hline 11000 \end{array}$	$\begin{array}{r} 111101 \\ + 101 \\ \hline 1000010 \end{array}$
---	--

- Note: $1 + 1 = 0$ with a carry 1

Two's Complement Representation (cont.)

- Now, we can easily compute the two's complement of binary numbers
 - **Example:** What is the two's complement of **1011**?
 - **Solution**
 - Step 1. Flip all bits to get **0100**
 - Step 2. Add 1 to the result to get **0100 + 1 = 0101**.
 - Therefore, the two's complement of **1011** is **0101**.
- **Remark:** The two's complement of a binary number must be expressed in the same bit pattern in order to get a correct interpretation of the result. See below.

Two's Complement Representation (cont.)

- **Example:** Given the nibble **0000**, find its two complement.
 - **Solution**
 - Step 1. Flip all bits to get **1111**
 - Step 2. Add 1 to the result to get **1111 + 1 = 10000**.
 - **Remark:** The left most carry bit is discarded because it goes out of the capacity of a nibble. Therefore, the two's complement of **0000** is **0000**.
- **Example:** Given the nibble **1111**, find its two complement.
 - **Solution**
 - Step 1. Flip all bits to get **0000**
 - Step 2. Add 1 to the result to get **0000 + 1 = 0001**.
 - Therefore, the two's complement of **1111** is **0001**.

Representation of Integers in Two's Complement

- How do we represent signed integers in two's complement?
- Solution

– Positive Decimal Numbers and Zero

Step 1. Their two's complement representation is the same as their unsigned binary representation

– Negative Decimal Numbers

Step 1. Find the unsigned binary representation of the decimal number without the negative sign

Step 2. Flip all the bits in Step 1

Step 3. Add 1 to the result in Step 2

Examples

- Find the binary representation of 53 in two's complement representation as a byte
 - Solution
 - The unsigned binary representation of 53 as a byte is **00110101**. This is also its two's complement representation.
- Find the binary representation of -53 in two's complement as a byte
 - Solution
 - **Step 1.** Find binary representation of 53. It is **00110101**
 - **Step2.** Flip all bits to get **11001010**
 - **Step 3.** Add 1 to the result to get **11001011**
- What about -103 in two's complement as a byte pattern?
 - Solution: Steps 1 through 3 will respectively give the results: **01100111**, **10011000**, and **10011001**. Therefore the answer is **10011001**

Examples (cont.)

- Add 53 with -53 in two's complement using a byte.

– Solution:

$$\begin{array}{r} 53 \text{ is } 00110101 \\ + -53 \text{ is } 11001011 \\ \hline 0 \quad \textcolor{red}{1}00000000 \end{array}$$

- Once again, the left most carry bit will be **discarded** as it will go **out of the capacity of a byte**. Therefore the answer will be **zero** as expected!

Closer Look at Two's Complement Representation

- Now, let us see which **signed** integers can be represented by a given bit pattern
- For this purpose, let us consider a nibble

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

Decimal	-1	-2	-3	-4	-5	-6	-7	-8
Binary	1111	1110	1101	1100	1011	1010	1001	1000

- Notice that the bit pattern **1000** could have been used for **+8** or **-8** equally right. However, computer scientists decided to assign it **-8** for reasons that will be evident soon.

Properties of Two's Complement Representation

1. The two's complement of **positive** decimal numbers and **zero** always start with a **0** bit. While that of **negative** decimal numbers start with a **1** bit
2. A nibble can represent the signed integers -8 to +7 in two's complement, as shown above. Similarly, a byte can represent signed integers -128 to +127 in two's complement
3. Generally, an **n**-bit pattern can represent the signed integers $-(2^{n-1})$ to $+(2^{n-1}-1)$

Conversion from Two's Complement Binary Representation to Decimal Numbers

- In order to convert a given binary number in two's complement to decimal, follow these steps:
 - If it starts with a **0**, then it is positive or zero integer.
 1. Convert it to decimal by expansion with powers of 2
 2. The resulting decimal is the required answer
 - Else if it starts with a **1**, then it is negative integer.
 1. Find the two's complement of the given binary number
 2. Convert it to decimal by expansion with powers of 2
 3. The negative of the resulting decimal is the required answer

Examples

- Find the integers represented by the following binary numbers represented in two's complement: **010101**, **11101101**, and **10000000**.
 - **010101** is positive. Expanding it results 21. Therefore the answer is 21.
 - **11101101** is negative. Its two complement is **00010011**. This binary corresponds to the decimal 19. Therefore the answer is -19.
 - **10000000** is negative. Its two complement is **10000000**. This binary corresponds to the decimal 128. Therefore the answer is -128.

Examples (cont.)

- Consider the decimal number 5 in Byte pattern represented in two's complement
 - Its binary representation is **00000101**
 - Its two's complement is **11111011** which is -5
- Now consider the decimal number -5 in byte pattern represented in two's complement
 - Its binary representation is **11111011**
 - Its two's complement is **00000101** which is 5
- Thus, if the two's complement representation of an integer x is y then the two's complement of $-x$ is given by the two's complement of y .

Arithmetic of Integers represented in Two's Complement Representation

- Let us perform the operations $3 + -5$, $-3 + 5$ and $-3 + 5$ in two's complement in Nibble and Byte forms:

$\begin{array}{r} 3 \quad 0011 \\ + -5 \quad 1011 \\ \hline -2 \quad 1110 \rightarrow -2 \end{array}$	$\begin{array}{r} -3 \quad 1101 \\ + 5 \quad 0101 \\ \hline 2 \quad \textcolor{red}{1}0010 \rightarrow 2 \end{array}$	$\begin{array}{r} -3 \quad 11111101 \\ + 5 \quad 00000101 \\ \hline 2 \quad \textcolor{red}{1}00000010 \rightarrow 2 \end{array}$
---	---	---

- The discarded carry bits are shown in red.
- When a carry over goes beyond the capacity of the bit pattern under consideration, then the operation is said to give rise to an **overflow**

Information Representation in Computers

- So far, we have seen how signed or unsigned numbers are represented in computers using unsigned binary representation, sign and magnitude representation, and two's complement representation
- But what about alphabets (a,b,c,...; A, B, C,...), digits (0, 1, 2,...), symbols (., :, ;, {, }, (, &, α , β , ...) etc
- Solution
 - These objects are encoded with an agreed code as unsigned binary numbers. **Ascii code** is used mostly. See <http://www.asciitable.com/>

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[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	[NEGATIVE 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	[ENG OF 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	[SUBSTITUTE]	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	[DEL]

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	Ł	224	E0	α
129	81	ü	161	A1	í	193	C1	ł	225	E1	β
130	82	é	162	A2	ó	194	C2	Ṭ	226	E2	Γ
131	83	â	163	A3	ú	195	C3	ṭ	227	E3	π
132	84	à	164	A4	ñ	196	C4	—	228	E4	Σ
133	85	ä	165	A5	Ñ	197	C5	†	229	E5	σ
134	86	ã	166	A6	*	198	C6	‡	230	E6	μ
135	87	ç	167	A7	°	199	C7	‡	231	E7	ι
136	88	ê	168	A8	¿	200	C8	‡	232	E8	φ
137	89	ë	169	A9	¬	201	C9	ƒ	233	E9	θ
138	8A	è	170	AA	¬	202	CA	±	234	EA	Ω
139	8B	ı	171	AB	½	203	CB	ƒ	235	EB	δ
140	8C	î	172	AC	¼	204	CC	ƒ	236	EC	∞
141	8D	ï	173	AD		205	CD	=	237	ED	φ
142	8E	Ä	174	AE	«	206	CE	±	238	EE	ε
143	8F	Å	175	AF	»	207	CF	±	239	EF	Ω
144	90	E	176	B0	⌘	208	D0	±	240	F0	≡
145	91	æ	177	B1	⌘	209	D1	ƒ	241	F1	±
146	92	Æ	178	B2	⌘	210	D2	Ṭ	242	F2	≥
147	93	ø	179	B3		211	D3	Ṭ	243	F3	≤
148	94	ö	180	B4	†	212	D4	Ö	244	F4	
149	95	ó	181	B5	‡	213	D5	ƒ	245	F5	
150	96	û	182	B6	‡	214	D6	ƒ	246	F6	+
151	97	ü	183	B7	‡	215	D7	‡	247	F7	≈
152	98	ý	184	B8	‡	216	D8	‡	248	F8	≈
153	99	Û	185	B9	‡	217	D9	‡	249	F9	·
154	9A	Ü	186	BA		218	DA	ƒ	250	FA	·
155	9B	ƒ	187	BB	‡	219	DB	■	251	FB	√
156	9C	£	188	BC	‡	220	DC	■	252	FC	π
157	9D	¥	189	BD	‡	221	DD	■	253	FD	±
158	9E	₤	190	BE	‡	222	DE	■	254	FE	■
159	9F	ƒ	191	BF	¬	223	DF	■	255	FF	

Information Representation in Computers (cont.)

- **Ascii Code**

- Stands for: **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange
- It uses 8-bit (byte) to represent most symbols used in English language (alphabets, punctuation marks, symbols, digits,...)
- **It uses unsigned binary representation in binary**

- In order to extend Ascii code for other than English language, **Unicode** (16-bit) is used
- Recently, **ISO code** (32-bit) is also used to represent even more symbols: everything in the world

Information Representation in Computers (cont.)

- Consider the message “Hello 123”. How do we represent it in the computer?

Symbol	Ascii Code (Decimal)	Ascii Code (8-bit Binary)
H	72	01001000
e	101	01100101
l	108	01101100
l	108	01101100
o	111	01101111
	32	00100000
1	49	00110001
2	50	00110010
3	51	00110011

- Therefore the bit sequence

010010000110010101101100011011000110111100100000001100010011001000110011 represents the message “Hello 123”

Putting It All Together

- All information stored and processed in a computer is represented with binary digits (bits)
- Sequence of binary bits can represent signed or unsigned integers, floating numbers, symbols, instructions,... BUT their interpretation schemes are different. Example
 - The binary bits sequence 11001010 may represent
 - The integer **202** (**unsigned binary representation**), or
 - The integer **-74** (**sign and magnitude representation**), or
 - The integer **-54** (**two's complement representation**), or
 - The character **L** (**Ascii code**)

Putting It All Together(cont.)

- How about floating numbers (numbers with decimal point; example 1.5, -56.3, 2.07, 25.0). How are they represented in a computer?
 - Such types of numbers are represented in a completely different way. Not discussed in this course!
- It is the responsibility of a programming language to keep track of what information a given sequence of bits represents (signed numbers, unsigned numbers, floating numbers, Ascii code for symbols, instructions,...)

Hexadecimal Number Notation

- While the binary number system is the fundamental working system in computers, it is very difficult to read and write for us humans
- We can however group 4-bits (nibbles) together and write them as hexadecimal digits for easier reading and writing
- Hexadecimal number is a number system of base 16
- The basic hexadecimal digits are
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F

Hexadecimal Number Notation (cont.)

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

- Therefore

(**1000****1011**)_{binary} becomes (8B)_{hex}

(**1110****1111**)_{binary} becomes (EF)_{hex}

(A75C)_{hex} becomes (**1010****0111****1010****1100**)_{binary}

The Memory Unit of a Computer

- Let us start with a given File sitting on the desktop of your computer. Assume the size of this File is 25KB (Kilo-Bytes)? What does it mean?
 - First of kilo is a metric prefix that stands for 2^{10} . Therefore the size of a given File is 25KB is equivalent to saying $25 * 2^{10} = 25,600$ Bytes. Since a Byte is 8 bits, this means that the File contains $25,600 * 8 = 204,800$ bits (0s and 1s) inside it.
- Similarly Mega stands for 2^{20} and Giga stands for 2^{30} .
- Next, consider the random access memory (RAM) of a computer. It is customary to say a given laptop has 512MB of RAM. This means its RAM is $512 * 2^{20}$ Bytes long. This means the RAM can store a maximum of $512 * 8 * 2^{20}$ bits at once.

Algorithms

- What is an algorithm?
 - Set of instructions to perform a certain task
- Formally
 - An algorithm is a sequence of unambiguous instructions in order to perform a certain task, i.e., for obtaining output for any valid input in a finite amount of time
- Algorithms can be expressed as:
 - Plain English: usually in a step-by-step fashion
 - Pseudocode: Similar to computer program but with no programming language jargon
 - Flowchart: Diagrammatic representation of step-by-step instructions to show how things work

Algorithms: Example 1

- Suppose a person has certain amount of dollars between 1 and 100 (both inclusive)
- Problem
 - Write an algorithm to guess the amount of dollars the person has by repeatedly picking numbers between 1 and 100. After each pick, we will ask the person if our pick is too large or too small and then adjust our next pick accordingly. We end the task once our pick exactly matches the amount of dollars the person has.

Algorithms: Example 1 (cont.)

- First Solution
 - Start picking from 1 and increment by 1 until the correct amount is guessed
 - This solution (algorithm) is ok. But it will be very slow. Assuming the person has 100 dollars, this algorithm will need 100 attempts to get the correct answer. Can we do better? **YES!**
- Second Solution
 - Pick the number halfway between 1 and 100, i.e., 50. If our pick is the correct amount then we stop, we are done. If too small, our next guess must be between 51 and 100. If too large, our next guess must be between 1 and 49. We continue picking the number halfway between smallest and largest possible values until we guess the number right

Algorithms: Example 1 (cont.)

- **Pseudocode form**

Step 1. Smallest and Largest represent the boundaries

Step 2. $\text{Guess} = 0.5 * (\text{Smallest} + \text{Largest})$

Step 3. **If** Guess equals AMOUNT

➔ Viola! We have found the number. Done. Game over. Finish.

Step 4. **ELSE If** Guess is TOO SMALL

➔ $\text{Smallest} = \text{Guess} + 1$

➔ Go to Step 2

Step 5. **ELSE**

➔ $\text{Largest} = \text{Guess} - 1$

➔ Go to Step 2

Algorithms: Example 1 (cont.)

- How many guesses does our solution require in the worst scenario to get the right answer?
 - Solution: **$\log_2 100 = 7$ guesses!**
- Hence the second algorithm is faster than the first algorithm which required 100 guesses to get the right answer in the worst scenario
- We say the second algorithm is **more efficient!**
- Algorithms can be more efficient for speed or memory considerations

Algorithms: Example 2

- Greatest Common Divisor (GCD)
- **Problem**
 - Given two positive integers M and N with $M > N$, give an algorithm to find the GCD of M and N
- **Algorithm1**
 - The GCD is always between 1 and N . Therefore start by guessing the GCD to be N and see if it divides both M and N . If it does, then we have guessed right. Done. If not, decrement the guess by 1 and repeat the process until the correct GCD is obtained.
 - This requires **N** division operations in the worst case

Algorithms: Example 2 (cont.)

- **Algorithm2: The Euclidean Algorithm**

Step 1. Make sure $M > N$

Step 2. Divide M by N and record the Remainder.

Step 3. If Remainder is 0

→ Done. The GCD is N

Step 4. Else

→ $M = N$

→ $N = \text{Remainder}$

→ Go to Step 2.

Algorithms: Example 2 (cont.)

- How many operations (step 2) does Algorithm2 perform in the worst scenario?
 - **Solution**: The number of division operations never exceeds 5 times the number of digits (in base-10) of the smaller number N . Not easy to prove and therefore no need to understand why this is so!
- Again, we say the **Euclidean Algorithm is more efficient** than our first Algorithm. In fact, the Euclidean Algorithm is the most elegant and most efficient algorithm for finding GCD

Algorithms: Example 2 (cont.)

- In order to appreciate the notion of efficiency of algorithms, let us try to compute the GCD of two big numbers.
 - Example: Find $\text{GCD}(32,452,843, 15,485,863)$
- These two numbers are primes. Therefore their GCD is 1. Algorithm1 would require to make 15,485,863 division operations before it gets the correct GCD.
- How about Algorithm2? Algorithm2 would proceed as follows:
 $\text{GCD}(32,452,843, 15,485,863) = \text{GCD}(15,485,863, 1,481,117) =$
 $\text{GCD}(1,481,117, 674,693) = \text{GCD}(674,693, 131,731) = \text{GCD}(131,731,$
 $16,038) = \text{GCD}(16,038, 3,427) = \text{GCD}(3,427, 2,330) = \text{GCD}(2,330,$
 $1,097) = \text{GCD}(1,097, 136) = \text{GCD}(136, 9) = \text{GCD}(9, 1) = 1.$ How many division operations? Only 11 division operations! Hence in this particular example, Algorithm2 is almost one and half million times faster than Algorithm1. Assuming each division operation would take 1 second to perform, Algorithm2 would require 11 seconds while Algorithm1 would require almost 6 months to complete!!!

Programming

- A way of communicating to a computer in order to perform certain task
- Just like human beings have languages to understand one another, a computer also has a language through which we may communicate to it
- Unfortunately, since a computer is made up of transistors, it can only understand bits!
- How can we then communicate to a computer in an English language like manner?

Programming (cont.)

- Computer hardware manufacturers first developed logic gates that can send and receive information and data as a sequence of bits to the transistors in a computer
- Next, functional units and Arithmetic Logic Units were developed to communicate with logic gates
- Next, Assembly language was developed to communicate with arithmetic logic units and functional units
- Next, high level languages (almost English like) were developed to communicate with Assembly Language
- Thus, we use such high level programming languages to develop applications to perform a required task

C++ Programming Language

- A high level computer programming language
- Extremely rich language supporting even assembly language like programming
- Written in plain English language
- Many development tools available: Microsoft Visual C++, Xcode, Dev Cpp, etc
- Freely available... Microsoft Visual C++ 2010 Express Edition

Program (Software) Development Process

- **Analyze the problem**
 - Identify what is the input, what is required to be done, and what is the output required
- **Devise an Algorithm**
 - Find efficient algorithm for solving the problem
- **Implement**
 - Write your algorithm in a programming language (coding)
- **Test**
 - Run your program to see if it is working as required
- **Maintenance**
 - Add, remove, modify your code with time for better performance