SEC204

Computer Architecture and Low Level Programming

Dr. Vasilios Kelefouras

Email: v.kelefouras@plymouth.ac.uk

Website:
https://www.plymouth.ac.uk/staff/vasilios-kelefouras

Date
23/09/2019

School of Computing
(University of Plymouth)

Outline

- Positional Numbering Systems
- Signed Integer Representation
- Floating Point Representation
- Character Codes

Basics (1)

- The bit is the most basic unit of information in a computer
  - Switching activity 0 or 1
- A Byte is a group of 8 bits
  - A byte is the smallest possible addressable unit of computer storage
  - The term, “addressable,” means that a particular byte can be retrieved according to its location in memory
- A word is a contiguous group of bytes, e.g., an integer uses 4 bytes
- Word sizes of 4 or 8 bytes are most common

Basics (2)

Kilo- (K) = 1 thousand = $\mathbf{10^3}$ and 2^{10}

Mega- (M) = 1 million = $\mathbf{10^6}$ and 2^{20}

Giga- (G) = 1 billion = $\mathbf{10^9}$ and 2^{30}

Tera- (T) = 1 trillion = $\mathbf{10^{12}}$ and 2^{40}

Peta- (P) = 1 quadrillion = $\mathbf{10^{15}}$ and 2^{50}

Exa- (E) = 1 quintillion = $\mathbf{10^{18}}$ and 2^{60}

Zetta- (Z) = 1 sextillion = $\mathbf{10^{21}}$ and 2^{70}

Yotta- (Y) = 1 septillion = $\mathbf{10^{24}}$ and 2^{80}

Normally, powers of 2 are used for measuring capacity

Milli- (m) = 1 thousandth = $\mathbf{10^{-3}}$

Micro- (μ) = 1 millionth = $\mathbf{10^{-6}}$

Nano- (n) = 1 billionth = $\mathbf{10^{-9}}$

Pico- (p) = 1 trillionth = $\mathbf{10^{-12}}$

Basics (3)

- Hertz = clock cycles per second (frequency)
  - 1MHz = 1,000,000Hz
  - Processor speeds are measured in MHz or GHz
- Byte = a unit of storage
  - 1KB = 2^{10} = 1024 Bytes
  - 1MB = 2^{20} = 1,048,576 Bytes
  - 1GB = 2^{30} = 1,099,511,627,776 Bytes
- Main memory (RAM) is measured in GB
- Disk storage is measured in GB for small systems, TB (2^{40}) for large systems

POSITIONAL NUMBERING SYSTEMS (1)

- Positional numbering systems are systems in which the placement of a digit in connection to its intrinsic value determines its actual meaning in a numeral string

- The organization of any computer depends considerably on how it represents numbers, characters, and control information
  - **There are several positional numbering systems such as Decimal, Binary, Octal, Hexadecimal etc**

- The positioning system is provided as a subscript, e.g., 14_{10}, 10101_{2}, 82_{16}

POSITIONAL NUMBERING SYSTEMS (2)

- Our decimal system is the base-10 system. It uses powers of 10 for each position in a number

- The binary system is also called the base-2 system

- The hexadecimal system is the base-16 system

- The Mayan and other Mesoamerican cultures used a number system based in a base-20 system

Decimal System

- **Decimal system**: Our well known and used system.
  - **It uses 10 different digits: 0,1,2,3,4,5,6,7,8,9**
  - Our decimal system is the base-10 system. It uses powers of 10 for each position in a number
  - For example, the decimal number 947 in powers of 10 is

947 =

$=9\times100 + 4\times10 + 7\times1 =$

$=9\times10^2 + 4\times10^1 + 7\times10^0$

  - $70216=7\times10000+0\times1000+2\times100+1\times10+6\times1=$

$=7\times10^4+0\times10^3+2\times10^2+1\times10^1+6\times10^0$

- The decimal number 3812.46 in powers of 10 is ($3\times10^3 + 8\times10^2 + 1\times10^1+2\times10^0+ 4\times10^{-1}+ 6\times10^{-2}$)

Binary System

- A binary number is a number expressed in the base-2 numeral system or binary numeral system, which uses only two symbols: typically 0 (zero) and 1 (one)
- The base is 2
- **2 different digits are used: 0,1**
- For example, $101_2 = 1\mathbf{x}2^2 + 0\mathbf{x}2^1 + 1\mathbf{x}2^0$

  $= 1\mathbf{x}4 + 0\mathbf{x}2 + 1\mathbf{x}1$

  $= 5_{10}$
- The binary number 11001 in powers of 2 is: $1x2^4 + 1x2^3 + 0x2^2 + 0x2^1 + 1x2^0 = 16 + 8 + 0 + 0 + 1 = 25_{10}$
- $1011.101_2 =$

  $= 1\mathbf{x}2^3 + 0\mathbf{x}2^2 + 1\mathbf{x}2^1 + 1\mathbf{x}2^0 + 1\mathbf{x}2^{-1} + 0\mathbf{x}2^{-2} + 1\mathbf{x}2^{-3} =$

  $= 1\mathbf{x}8 + 0\mathbf{x}4 + 1\mathbf{x}2 + 1\mathbf{x}1 + 1\mathbf{x}0.5 + 0\mathbf{x}0.25 + 1\mathbf{x}0.125$

  $= 11.625_{10}$

Octal system

- The base is 8
- **8 different digits are used only: 0,1,2,3,4,5,6,7**
- For example: $436_8 = 4\mathbf{x}8^2 + 3\mathbf{x}8^1 + 6\mathbf{x}8^0$

$$= 4\mathbf{x}64 + 3\mathbf{x}8 + 6\mathbf{x}1$$

$$= 286_{10}$$

Convert the following octal number 205.24_8, to decimal:

$$205.24_8 = 2x8^2 + 0\mathbf{x}8^1 + 5\mathbf{x}8^0 + 2x8^{-1} + 4x8^{-2}$$

$$= 2x64 + 0 + 5 + 2x0.125 + 4x0.015625$$

$$= \mathbf{133.3125_{10}}$$

Hexadecimal system

- The base is 16
- **16 different digits are used: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F**

(we do not use numbers with 2 digits like 10,11,12,...), but
A instead of 10, B instead of 11, C instead of 12, etc)

- Example: $3B1_{16} = 3\mathbf{x}16^2 + 11\mathbf{x}16^1 + 1\mathbf{x}16^0$

$= 3\mathbf{x}256 + 11\mathbf{x}16 + 1 =$

$= 768 + 176 + 1 =$

$= 945_{10}$

Convert the following hexadecimal number $20C.2_{16}$ to decimal

$\mathbf{20C.2_{16}} = 2x16^2 + 0x16^1 + 12x16^0 + 2x16^{-1} =$

$= 2x256 + 0 + 12x1 + 2x0.0625 =$

$= 512 + 12 + 0.125 =$

$= \mathbf{524.125_{10}}$

In the Lab session

- You will learn how to convert from a system to another…

Positional Numbering Systems - General case

- Base: r
- **Uses r different digits: 0,1,2,3,..r-1**

- $N_r = A_{n-1} A_{n-2} \ldots A_1 A_0 A_{-1} A_{-2} \ldots A_{-(m-1)} A_{-m}$

  $N_r = A_{n-1} \times r^{n-1} + A_{n-2} \times r^{n-2} + \ldots + A_1 \times r^1 + A_0 \times r^0 + A_{-1} \times r^{-1} + A_{-2} \times r^{-2} + \ldots + A_{-(m-1)} \times r^{-(m-1)} + A_{-m} \times r^{-m}$

To better understand the above formula consider that if $234.03_5 = ?_{10}$ then n=3, m=2 and r=5

- The left most digit (A**n-1**) is called Most Significant Bit-(MSB) while the right most (A**-m**) Least Significant Bit-(LSB)

Basic arithmetic operations

- The basic arithmetic operations are applied to **all** the previous numerical systems. There are:
  - Addition
  - Subtraction
  - Multiplication
  - Division

- **Examples are provided in the lab session…**

Signed integer representation

Introduction

- In practice we have to use negative binary numbers too. **We need to define signed binary numbers**

- ✓ **There are three ways in which signed binary integers may be expressed:**

1. **Signed magnitude**

2. **One's complement**
3. **Two's complement**

Signed Magnitude Representation (1)

- **Allocate the high-order (leftmost) bit to indicate the sign of a number**
  - The high-order bit is the leftmost bit. It is also called the most significant bit
  - 0 is used to indicate a positive number; 1 indicates a negative number
- The remaining bits contain the value of the number
- Note that we also **pay attention to the number of bits used** to represent signed binary numbers
  - i.e. if using 4 bit numbers, then we use 0001_2 rather than 1_2
- In an 8-bit word, signed magnitude representation places the absolute value of the number in the 7 bits to the right of the sign bit

For example:

+3 is: **0**0000011

- 3 is: **1**0000011

Signed Magnitude Representation (2)

- The "binary addition algorithm" does NOT work with sign-magnitude

$0 \quad 0\,1\,1_2 = 3_{10}$

$1 \quad 1\,0\,0_2 = -4_{10}$

$$\begin{array}{lr} 0 & 0\,1\,1 \\ 1 & +\ 1\,0\,0 \\ \hline 1 & 1\,1\,1 \end{array}$$

this is wrong

Signed Magnitude: intuitive for humans, difficult for computers

- Signed magnitude representation is easy for people to understand, but it requires complicated computer hardware

- Also it allows two different representations for zero: positive zero and negative zero

- As such, computer systems employ **complement systems** for signed number representation

Signed Integer Representation
Complement Systems

- In binary systems, these are:
  - **One's Complement.** To represent **negative** values, invert all the bits in the binary representation of the number (swapping 0s for 1s and vice versa)
    - 1 becomes 0 and 0 becomes 1
    - To represent **positive** numbers no change is applied

    For example, using 8-bit one's complement representation

    + 3 is: 00000011

    - 3 is: 11111100

    More examples

    X=11011100, 1C(X)=00100011

    X=1011, 1C(X)=?

    - One's complement still has the disadvantage of having two different representations for zero: positive zero and negative zero
    - In addition positive and negative integers need to be processed separately
    - Two's complement solves this problem
  - **Two's complement**
    - One's Complement add 1

Signed Integer Representation
Two's Complement

Two's complement 2C(X)

- You represent **positive** numbers, just like the unsigned numbers
- To represent **negative values**, start with the corresponding positive number, invert all the bits. Then add 1

- For example, using 8-bit two's complement representation:

+ 3 is: 00000011 *1.Start with positive number*

11111100 *2.Invert bits*

\+ 1 *3. Add 1*

- 3 is: 11111101

-3 in 8-bit Two's Complement Representation is 11111101

✓Negative numbers must always start with '1'
✓Both positive and negative numbers must have the same number of bits

Floating-Point Representation (1)

- To represent real numbers with fractional values, floating-point representation is used
- Floating-point numbers are often expressed in scientific notation
  - For example: $0.125 = 1.25 \times 10^{-1}$

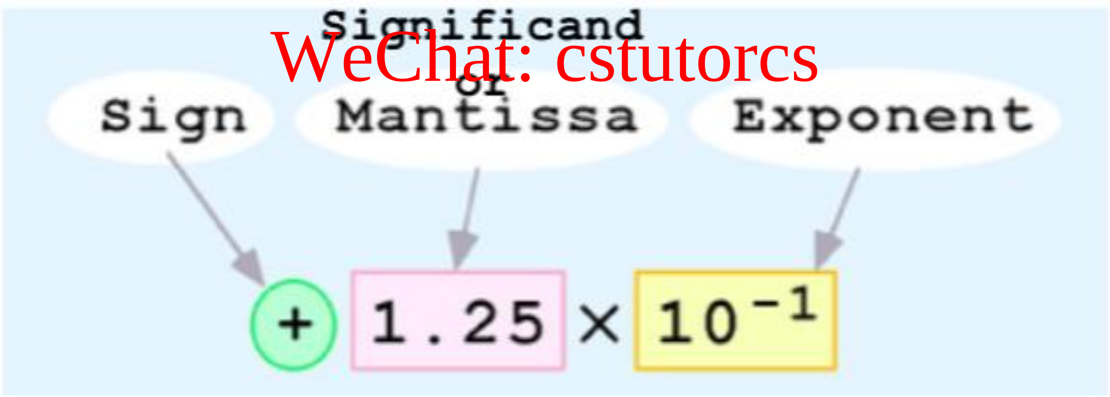
- Remember that when a number is **multiplied by its base**, e.g., 10, then we add a zero or we move the ',' by one position to the right

  - 235x10 = 2350

  - 1.345x10=13.45
  - 110_2x2 = 1100_2 *($6x2=12_{10}$)*
  - 101.11_2x2=1011.1 *($5.75x2=11.5_{10}$)*

Floating-Point Representation (2)

- Computers use a form of scientific notation for floating-point representation
  - Single Precision floating point format 32-bit
  - Double Precision floating point format 64-bit
- Numbers written in scientific notation have three components:



Single precision Floating-Point format (1)

A binary number is represented in FP format as follows:

1. We write the number using only a single non-zero digit before the radix point :

   e.g., $1011010010001 = 1.011010010001 \times 2^{12}$

   $1101.10111 = 1.10110111 \times 2^{3}$

2. Then we transform the number to the following format using 32 bits

$$N = (-1)^S (1+F)(2^{E-127})$$

Sign-S 1-bit	Exponent-E 8 - bits	Mantissa (Fraction) - F 23 - bits

S: Sign, 0/1 for positives/negatives, respectively

E: Exponent. E-127=exp, where exp is the corresponding exponent

F: Significant or Mantissa. We write the fractional part in 23 bits

E=127+exp in order to avoid using negative numbers. exp=[-127,128] and therefore E=[0,255] – 255 needs 8 bits

Single precision Floating-Point format (2)

Convert the positive number N=1011010010001 in Floating point format

Step1: 1011010010001= 1.011010010001 x 2^{12}

Step2: **$N = (-1)^S (1+F)(2^{E-127})$**

S = 0 (positive number)

E - 127 = 12, and thus E = 139_{10} and E = 10001011_2

F = 01101001000100000000000

Therefore N in FP format is:

0	10001011	01101001000100000000000

Single precision Floating-Point format (3)

Suppose that the 32-bit floating-point representation pattern is the following. Find the binary number

1	10010001	10001110001000000000000

S is 1 and thus the number is negative

E is 10010001 = 145_{10}, and thus the exponent is exp=E-127=145-127=18

F= 10001110001000000000000

$N = (-1)^{S}(1+F)(2^{E-127})$

N is $(-1)^{1}$ x 1.10001110001000000000000x2^{18} or

N = - 1100011100010000000

Floating-Point Representation (1)

- No matter how many bits we use in a FP representation, the model is finite
  - The real number system is, of course, infinite, so our models can give nothing more than an approximation of a real value
  - e.g., how to represent 33.33333333333333333333?
- At some point, every model breaks down, introducing errors into our calculations
  - By using a greater number of bits in our model, we can reduce these errors, but we can never totally eliminate them

Why is 0.1+0.2 not equal to 0.3 in most programming languages?

- computers use a binary floating point format that cannot accurately represent a number like 0.1_{10}
- 0.1_{10} is already rounded to the nearest number in that format
- 0.1_{10} doesn't exist in the FP representation
- 0.1_{10} is already rounded to the nearest number in that format, which results in a small rounding error
- This means that 0.1_{10} is converted to a binary number that's just very close to 0.1_{10}
- The error is tiny since 0.1_{10} is 0.1000000000000000055511151231257827
- The constants 0.2_{10} and 0.3_{10} are also approximations to their true values
- So, $0.1_{10} + 0.2_{10} == 0.300000000000000044408920985006_{10}$

Character Codes

- So far, we have learnt how to represent numbers. How about text?
- To represent text characters, we use character codes
  - Essentially, we assign a number for each character we want to represent

- As computers have evolved, character codes have evolved. Larger computer memories and storage devices permit richer character codes

- Some of the character codes are
  1. BCD
  2. ASCII (American Standard Code for Information Interchange) (7 bits)
  3. Extended ASCII (8-bits)
  4. Unicode
  5. and others

- A binary number of n bits gives 2^n different codes
  - For n=2 there are $2^2=4$ different codes, i.e., bit combinations {00, 01, 10, 11}

Binary Coded Decimal (BCD) code

- when numbers, letters or words are represented by a specific group of symbols, it is said that the number, letter or word is being encoded. The group of symbols is called as a code

- **Binary Coded Decimal (BCD) code**
  - In this code each decimal digit is represented by a 4-bit binary number
  - BCD is a way to express each of the decimal digits with a binary code
  - In the BCD, with four bits we can represent sixteen numbers (0000 to 1111)

$256_{10} = 0010\ 0101\ 0110_{BCD}$

And vise versa

$0011\ 1000\ 1001_{BCD} = 389_{10}$

ASCII Code

- The most widely accepted code is called the American Standard Code for Information Interchange (ASCII).
- The ASCII code associates an integer value for each symbol in the character set, such as letters, digits, punctuation marks, special characters, and control characters

- The ASCII table has 128 characters, with values from 0 through 127. Thus, 7 bits are sufficient to represent a character in ASCII

ASCII Code

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Extended ASCII Characters

- ASCII was designed in the 1960s for teleprinters and telegraphy, and some computing
- The number of printable characters was deliberately kept small, to keep teleprinters and line printers inexpensive
- When computers and peripherals standardized on eight-bit bytes, it became obvious that computers and software could handle text that uses 256-character sets at almost no additional cost in programming, and no additional cost for storage
- An eight-bit character set (using one byte per character) encodes 256 characters, so it can include ASCII plus 128 more characters
- The extra characters represent characters from foreign languages and special symbols for drawing pictures

A set of codes that extends the basic ASCII set. The extended ASCII character set uses 8 bits, which gives it an additional 128 characters

128	Ç	144	É	160	á	176	░	192	└	208	╨	224	α	240	≡
129	ü	145	æ	161	í	177	▒	193	┴	209	╤	225	ß	241	±
130	é	146	Æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
131	â	147	ô	163	ú	179	│	195	├	211	╙	227	π	243	≤
132	ä	148	ö	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
133	à	149	ò	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
134	å	150	û	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷
135	ç	151	ù	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
136	ê	152	ÿ	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
137	ë	153	Ö	169	⌐	185	╣	201	╔	217	┘	233	Θ	249	∙
138	è	154	Ü	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
139	ï	155	¢	171	½	187	╗	203	╦	219	█	235	δ	251	√
140	î	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	^{n}
141	ì	157	¥	173	¡	189	╜	205	═	221	▌	237	φ	253	^{2}
142	Ä	158	₧	174	«	190	╛	206	╬	222	▐	238	ε	254	■
143	Å	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	

Source: www.LookupTables.com

UNICODE

- Many of today's systems embrace Unicode that can encode the characters of every language in the world
  - The Java programming language, and some operating systems now use Unicode as their default character code
    - UTF-8 (8-bits: essentially the extended ASCII Table)
    - UTF-16 (16 bits: Most spoken languages in the world, widely used)
    - UTF-32 (32 bits: includes past languages, space inefficient)

Any questions?