x86 Assembly Programming

Lesson 1 – Data Representation

Ludvik Jerabek

Binary to Decimal

Base 2 (binary) numbers are represented using the digits 0 and 1. Binary numbers are like Base 10 (decimal) numbers except that the positions from (right to left) increase by powers of 2, instead of powers of 10. Decimal numbers have a 1's place, 10's place, 100's place, 1000's place, 10000's place, and so on increasing in powers of 10. Decimal number place values are calculated via 10^n starting at n=0 (far right). Similarly binary place values are calculated 2^n starting at n=0 (far right). Decimal numbers unlike binary can have more digits 0-9 in each place eg. 9 in the 10's place is 90 or 9*(10^1). Thus digit*(base^digit_position) = value. All place values are then added together to assemble the final decimal number. Eg. 10101101 = 128 + 32 + 8 + 4 + 2 + 1 = 175 see below:

Digit Position	7	6	5	4	3	2	1	0	Total
Base 10 (10^n)	10^7	10^6	10^5	10^4	10^3	10^2	10^1	10^0	-
Base 10 n's place	10,000,000's	1,000,000's	100,000's	10,000's	1,000's	100's	10's	1's	-
Decimal Number Position Values 53,125,101	5*(10^7)	3*(10^6)	1*(10^5)	2*(10^4)	5*(10^3)	1*(10^2)	0*(10^1)	1*(10^0)	53,125,101
53,125,101 Add Position Values	50,000,000	3,000,000	100,000	20,000	5,000	100	0	1	53,125,101
Base 2 (2^n)	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	-
Base 2 n's place	128's	64's	32's	16's	8's	4's	2's	1's	-
Binary Number Position Values 10101101	1*(2^7)	0*(2^6)	1*(2^5)	0*(2^4)	1*(2^3)	1*(2^2)	1*(2^1)	1*(2^0)	175
10101100 = 175 Add Position Values	128	0	32	0	8	4	2	1	175

Binary and Hexadecimal Numbers

- Base 2 – Binary Numbers eg. 11110001
  - Binary numbers are long and hard to read so hexadecimal (hex) numbers are used to represent binary numbers in a more readable format. A byte is 8 bits, a nibble is 4 bits, and each bit is a binary 1 or 0.
- Base 16 – Hexadecimal Numbers eg. 0xF1
  - Hex numbers range from 0-9 followed by A-F representing 10-15 respectively. Every two hex digits represent a byte of data eg. 0xFF is equal to an unsigned decimal value 255. A single hex digit represents a nibble (half byte or 4 bits). Thus hex 0-F can be represented in binary as 0000-1111 or decimal as 0-15. This will come in very handy.

Decimal, Hex, and Binary

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

To prevent confusion with decimal numbers hex numbers should be prepended with 0x

Converting Binary to Hex

- Take a long binary number and break it apart at each nibble starting on the right to the left.

(16 bits, 2 bytes)	1011010010110101
(Split into nibbles)	1011 0100 1011 0101
(Nibbles to Hex)	B 4 B 5
(Final Hex Number)	0xB4B5

Converting Binary to Hex

- Take each hex digit and replace them using the hex to binary table on (slide 3).

(Hex)	0xABCD
(Lookup A)	1010 = A
(Lookup B)	1011 = B
(Lookup C)	1100 = C
(Lookup D)	1101 = D
	A B C D
(Replace ABCD)	1010 1011 1100 1101
(Binary)	1010101111001101

Converting Hex to Binary

- It is possible that you may not have a binary number which consists of a perfect 4 bits or nibble.

(3bits)	101
(Hex)	0x5
(14 bits)	11010010110101
(Split into nibbles)	11 0100 1011 0101
(Hex)	3 4 B 5
(Final Hex Number)	0x34B5

Converting Decimal to Base N

- Divide decimal number by Base, get the quotient and remainder.
  - Remainder is the next digit (right to left)
  - Quotient is the next decimal for division
  - Do until decimal number is 0
- Decimal 254 to Base 16 example:

$254 \div 16$ = Quotient = 15, Remainder = 14 = 0xE

Right most number E

$15 \div 16$ = Quotient = 0 , Remainder = 15 = 0xF

0xFE = 254

Converting Decimal to Base N

- Decimal 64 to Base 2 example:

  $64 \div 2$ = Quotient = 32, Remainder = 0 Right most

  $32 \div 2$ = Quotient = 16 , Remainder = 0

  $16 \div 2$ = Quotient = 8 , Remainder = 0

  $8 \div 2$ = Quotient = 4 , Remainder = 0

  $4 \div 2$ = Quotient = 2 , Remainder = 0

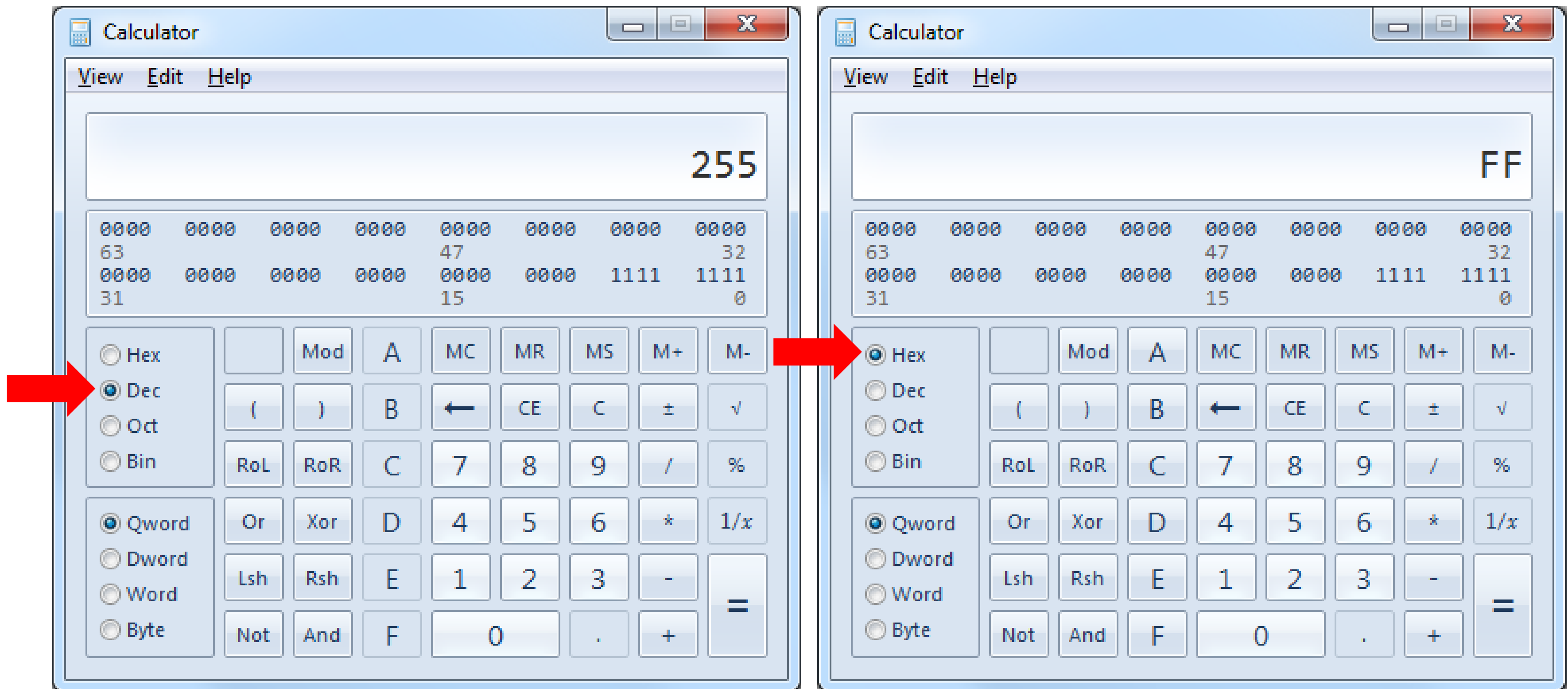
  $2 \div 2$ = Quotient = 1 , Remainder = 0

  $1 \div 2$ = Quotient = 0 , Remainder = 1 Left most

  (Binary) 1000000

Number Conversion Tools

- Windows Calculator in Programming Mode
  - Enter number in decimal mode then toggle mode to Hex, Oct, or Bin.



Character Codes - ASCII

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	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	Neg. 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	End 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	Substitution	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	□

- Letters, numbers, punctuation marks, and special symbols are numeric values which result in the associated character being displayed.
- This character set is known as the ASCII character set.
- The ASCII system uses 7 bits to represent decimal 0-127 or hex 0x00-0x7F

Character Codes – Ext. ASCII

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	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	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	□

- Additionally there are several versions of the extended ASCII characters decimal 128-255 or hex 0x80-0xFF.
- The 8 bit limitation for ASCII characters lead to the creation of 16 bit characters called Unicode.
- Unicode provides 65535 different characters in order to support different languages eg. Chinese.
- The first 128 characters of Unicode are comprised of the ASCII character set except in 16 bits eg. 0x0000-0x007F

Example of ASCII Strings

- Using the ASCII table on page 11 we can find the hex numbers for a string.
  - This is fun!
  - 0x54 = T
  - 0x68 = h
  - 0x69 = i
  - 0x73 = s
  - 0x20 = <space>
  - 0x69 = i
  - 0x73 = s
  - 0x20 = <space>
  - 0x66 = f
  - 0x75 = u
  - 0x6E = n
  - 0x21 = !
- So in hex 0x54 0x68 0x69 0x73 0x20 0x69 0x73 0x20 0x66 0x75 0x6E 0x21

Try it!

Try your best to do it manually

Decimal	Hex	Binary	ASCII
			NULL
			?
			Z
			d
		100	
		1000101	
		1010	
		111101010010	
	0xFFFF		
	0x10		
	0x16		
	0x7C8		
100			
256			
4253			
19200			

Adding and Subtracting in Binary

- Same as decimal; if sum of digits in a given position exceeds the base (10 for decimal, 2 for binary) then there is a carry into the next higher position

	1	
	3	9
+	3	5
	7	4

				1	1	1	1	
	0	0	0	0	1	0	1	1
+	0	0	0	0	0	1	0	1
	0	0	0	1	0	0	0	0

Overflow Rules

Overflow can only happen when "adding" two numbers of the same sign and getting a different sign. So, to detect overflow we don't care about any bits except the sign bits. Ignore the other bits.

Carry Out	Sign Bit							
	0	0	0	0	1	0	1	0
+	0	0	0	0	0	1	0	1
	0	0	0	0	1	1	1	1

Carry into sign bit?	Carry out of sign bit?	Overflow?
No	No	No

Overflow Rules

Overflow can only happen when "adding" two numbers of the same sign and getting a different sign. In the case below we add two negative numbers (both have 1 in the sign bit position) and it results in a positive result 00001111.

Carry Out	Sign Bit						
1							
	1	0	0	0	1	0	1	0
+	1	0	0	0	0	1	0	1
1	0	0	0	0	1	1	1	1

Carry into sign bit?	Carry out of sign bit?	Overflow?
No	Yes	Yes

Overflow Rules

Overflow can only happen when "adding" two numbers of the same sign and getting a different sign. In the case below both numbers are positive and result in a carry into the sign bit changing the sign of the value from positive to negative.

Carry Out	Sign Bit						
	1						
	0	1	0	0	1	0	1	0
+	0	1	0	0	0	1	0	1
	1	0	0	0	1	1	1	1

Carry into sign bit?	Carry out of sign bit?	Overflow?
Yes	No	Yes

Overflow Rules

Overflow can only happen when "adding" two numbers of the same sign and getting a different sign. In this case the top number is a positive number and the bottom is a negative number. This means there will not be overflow since overflow can only occur if both numbers are of the the same sign.

Carry Out	Sign Bit							
1	1							
	0	1	0	0	1	0	1	0
+	1	1	0	0	0	1	0	1
1	0	0	0	0	1	1	1	1

Carry into sign bit?	Carry out of sign bit?	Overflow?
Yes	Yes	No

Bitwise - NOT(¬)

- Truth table for NOT(¬) logic

A	¬A
0	1
1	0

Bitwise - AND(^)

- Truth table for AND(^) logic
  - A ^ B is 1 if both A and B are 1

A	B	A^B
0	0	0
1	0	0
0	1	0
1	1	1

Bitwise - OR(v)

- Truth table for OR(v) logic
  - A v B is 1 if either A or B is 1

A	B	AvB
0	0	0
1	0	1
0	1	1
1	1	1

Bitwise - XOR(⊕)

- Truth table for XOR(⊕) logic
  - Sometimes called add without a carry

A	B	A⊕B
0	0	0
1	0	1
0	1	1
1	1	0

More Bitwise Fun

AND(^) of two binary numbers

	1	1	1	0	0	0	0	1
^	0	0	0	0	0	0	0	1
	0	0	0	0	0	0	0	1

OR(v) of two binary numbers

	1	1	1	0	0	0	0	1
v	0	0	0	0	0	0	0	1
	1	1	1	0	0	0	0	1

XOR(⊕) of two binary numbers

	1	1	1	0	0	0	0	1
⊕	0	0	0	0	0	0	0	1
	1	1	1	0	0	0	0	0

1's Complement or Logical NOT

- 1's Complement is also known as a bit complement or a logical not. Take every bit and write the opposite bit. The character ¬ denotes a logical NOT.

¬	1	0	1	0	1	0	1	0
	0	1	0	1	0	1	0	1

Signed and Unsigned Numbers

- How do we represent negative numbers on a computer system?
- We could try ASCII and store each number as ASCII in memory and use “–” to designate it as negative eg: -123
- This would require 4 bytes to store. Thus it would require 32 bits of data to represent a number that is not even a byte in size, remember 1 byte (8bits) can represent 0-255 in decimal.
- ASCII number representation would be inefficient in two ways
  - It wastes space
  - Difficult to manipulate add, subtract, multiply, divide since it essentially is a string and not a number.
- This means there needs to be a better and more efficient way to deal with signed and unsigned numbers.
  - The **2’s complement system** is the answer

2’s Complement System

- In the system numbers are of a fixed length
  - BYTE (8bits)
  - WORD (16bits)
  - DWORD (32bits)
  - QWORD (64bits)
- Positive numbers most significant bit is 0
  - MAX BYTE = 01111111 = 127
- Negative numbers most significant bit is 1
  - MAX NEG BYTE = 10000000 = -128

Calculating the 2’s Complement

- Let’s say you want to calculate -30 start with positive 30 in binary 8 (bits)

Logical NOT (¬) the number 30 = 00011110	¬	0	0	0	1	1	1	1	0
-31 or unsigned decimal 225 = 11100001		1	1	1	0	0	0	0	1

-31 or unsigned decimal 225 = 11100001		1	1	1	0	0	0	0	1
Add 1	+	0	0	0	0	0	0	0	1
2’s Complement of 30 is unsigned 226 or -30		1	1	1	0	0	0	1	0

- 11100010 is the twos complement of 30 and it’s value is -30.
- Note the unsigned value of 11100010 is decimal 226 so it’s not possible to simple get the decimal value of the binary. In order to determine the value of the negative number a 2’s complement must be done.

Calculating the 2's Complement

- In order to determine what a negative number is you must take the 2's complement

-30 or unsigned 226 = 11100010	¬	1	1	1	0	0	0	1	0
¬-30 or unsigned 226 = 00011101 = 29		0	0	0	1	1	1	0	1

29 = 00011101		0	0	0	1	1	1	0	1
Add 1	+	0	0	0	0	0	0	0	1
2's Complement of -30 is 30		0	0	0	1	1	1	1	0

- So in order to determine the value of a negative number use twos complement

Floating Point Numbers

- Floating Point Numbers are stored in binary via IEEE 754
- IEEE 754 floating point numbers have three basic bit groups: sign, exponent, and mantissa. The floating point values are stored in 32bits and 64 bits. The below table highlights the number of bits and position of the bits.

	Sign	Exponent	Fraction	Bias
32 bit (Single Precision)	1 [31]	8 [30-23]	23 [22-00]	127
64 bit (Double Precision)	1 [63]	11 [62-52]	52 [51-00]	1023

- The standard is outside of the scope of this document for more information on the IEEE 754 please see: http://en.wikipedia.org/wiki/IEEE_floating_point

Try it!

Try your best to do it manually

Solve	Operation	Binary Result
¬01010010	NOT(¬)	
00000001●11111110	AND(●)	
00000001+11111110	OR(+)	
10101010⊕10101011	XOR(⊕)	
-5	Using 2's Complement	