CMSC 313
COMPUTER ORGANIZATION
&
ASSEMBLY LANGUAGE PROGRAMMING

LECTURE 02, SPRING 2013

TOPICS TODAY

- Bits of Memory
- Data formats for negative numbers
- Modulo arithmetic & two’s complement
- Floating point formats (briefly)
- Characters & strings

BITS OF MEMORY

Random Access Memory (RAM)

- A single byte of memory holds 8 binary digits (bits).
- Each byte of memory has its own address.
- A 32-bit CPU can address 4 gigabytes of memory, but a machine may have much less (e.g., 256MB).
- For now, think of RAM as one big array of bytes.
- The data stored in a byte of memory is not typed.
- The assembly language programmer must remember whether the data stored in a byte is a character, an unsigned number, a signed number, part of a multi-byte number, ...

Common Sizes for Data Types

- A byte is composed of 8 bits. Two nibbles make up a byte.
- Halfwords, words, doublewords, and quadwords are composed of bytes as shown below:

Bit	0
Nibble	0110
Byte	10110000
16-bit word (halfword)	11001001 01000110
32-bit word	10110100 00110101 10011001 01011000
64-bit word (double)	01011000 01010101 10110000 11110011
	11001110 11101110 01111000 00110101
128-bit word (quad)	01011000 01010101 10110000 11110011
	11001110 11101110 01111000 00110101
	00001011 10100110 11110010 11100110
	10100100 01000100 10100101 01010001

5.2 Instruction Formats

- Byte ordering, or *endianness*, is another major architectural consideration.
- If we have a two-byte integer, the integer may be stored so that the least significant byte is followed by the most significant byte or vice versa.
  - In *little endian* machines, the least significant byte is followed by the most significant byte.
  - *Big endian* machines store the most significant byte first (at the lower address).

5.2 Instruction Formats

- As an example, suppose we have the hexadecimal number 12345678.
- The big endian and small endian arrangements of the bytes are shown below.

Address →	00	01	10	11
Big Endian	12	34	56	78
Little Endian	78	56	34	12

5.2 Instruction Formats

- Big endian:
  - Is more natural.
  - The sign of the number can be determined by looking at the byte at address offset 0.
  - Strings and integers are stored in the same order.
- Little endian:
  - Makes it easier to place values on non-word boundaries.
  - Conversion from a 16-bit integer address to a 32-bit integer address does not require any arithmetic.

NEGATIVE NUMBERS

SIGNED INTEGER FORMATS

- Signed magnitude
- One’s complement
- Two’s complement
- Excess (biased)

SIGNED MAGNITUDE

- Store sign in leftmost bit, 1 = negative
- Example (8-bits):

```
 37 = 0010 0101
-37 = 1010 0101
```

ONE'S COMPLEMENT

- Negate by *flipping* each bit
- Example (8-bits):

```
 37 = 0010 0101
-37 = 1101 1010
```

TWO'S COMPLEMENT

- Negate by flipping each bit and adding 1
- Example (8-bits):

```
37  =  0010  0101

       1101  1010
     +          1
     ------------
       1101  1011  =  -37
```

EXCESS (BIASED)

- Add bias to two’s complement
- Example (8-bit excess 128):

```
37  =  0010 0101
       1101 1010
      +         1
      -----------
       1101 1011
      +1000 0000
      -----------
       0101 1011  =  -37
```

Example: Convert -123

- Signed Magnitude

  $123_{10} = 64 + 32 + 16 + 8 + 2 + 1 = 0111\ 1011_2$
  $-123_{10} \Rightarrow 1111\ 1011_2$

- One's Complement (flip the bits)

  $-123_{10} \Rightarrow 1000\ 0100_2$

- Two's Complement (add 1 to one's complement)

  $-123_{10} \Rightarrow 1000\ 0101_2$

- Excess 128 (add 128 to two's complement)

  $-123_{10} \Rightarrow 0000\ 0101_2$

PICKING A FORMAT

How do you

- check for negative numbers?
- test if a number is zero?
- add & subtract positive & negative numbers?
- determine if an overflow has occurred?
- check if one number is larger than another?

Implemented in hardware: simpler = better

3-bit Signed Integer Representations

Decimal	Unsigned	Sign Mag	1's Comp	2's Comp	Excess 4
7	111				
6	110				
5	101				
4	100				
3	011	011	011	011	111
2	010	010	010	010	110
1	001	001	001	001	101
0	000	000/100	000/111	000	100
-1		101	110	111	011
-2		110	101	110	010
-3		111	100	101	001
-4				100	000

2.4 Signed Integer Representation

- Binary addition is as easy as it gets. You need to know only four rules:

```
  0 + 0 =  0      0 + 1 =  1
  1 + 0 =  1      1 + 1 = 10
```

- The simplicity of this system makes it possible for digital circuits to carry out arithmetic operations.
  - We will describe these circuits in Chapter 3.

Let's see how the addition rules work with signed magnitude numbers . . .

2.4 Signed Integer Representation

- Example:
  - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- First, convert 75 and 46 to binary, and arrange as a sum, but separate the (positive) sign bits from the magnitude bits.

$$\begin{array}{cr} 0 & 1001011 \\ 0 & \underline{+\ 0101110} \end{array}$$

2.4 Signed Integer Representation

- Example:
  - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- Once we have worked our way through all eight bits, we are done.

```
          1 1 1
0     1 0 0 1 0 1 1
0 +   0 1 0 1 1 1 0
    ---------------
0     1 1 1 1 0 0 1
```

In this example, we were careful to pick two values whose sum would fit into seven bits. If that is not the case, we have a problem.

2.4 Signed Integer Representation

- Example:
  - Using signed magnitude binary arithmetic, find the sum of 107 and 46.
- We see that the carry from the seventh bit *overflows* and is discarded, giving us the erroneous result: 107 + 46 = 25.

$$
\begin{array}{ccccccccc}
\boxed{1} \longleftarrow & & & 1 & & 1 & 1 & 1 & & \\
 & 0 & & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\
 & 0 & + & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ \hline
 & 0 & & 0 & 0 & 1 & 1 & 0 & 0 & 1
\end{array}
$$

2.4 Signed Integer Representation

- The signs in signed magnitude representation work just like the signs in pencil and paper arithmetic.
  - Example: Using signed magnitude binary arithmetic, find the sum of - 46 and - 25.

```
     11
1   0101110
1 + 0011001
  ---------
1   1000111
```

- Because the signs are the same, all we do is add the numbers and supply the negative sign when we are done.

2.4 Signed Integer Representation

- Mixed sign addition (or subtraction) is done the same way.
  - Example: Using signed magnitude binary arithmetic, find the sum of 46 and - 25.

$$\begin{array}{ccccccccc}
0 & & 0 & \overset{0}{\cancel{1}} & \overset{2}{0} & 1 & 1 & \overset{0}{\cancel{1}} & \overset{2}{0} \\
1 & + & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
\hline
0 & & 0 & 0 & 1 & 0 & 1 & 0 & 1
\end{array}$$

- The sign of the result gets the sign of the number that is larger.
  - Note the "borrows" from the second and sixth bits.

2.4 Signed Integer Representation

- Signed magnitude representation is easy for people to understand, but it requires complicated computer hardware.
- Another disadvantage of signed magnitude is that it allows two different representations for zero: positive zero and negative zero.
- For these reasons (among others) computers systems employ *complement systems* for numeric value representation.

8-bit Two's Complement Addition

```
     54₁₀ = 0011 0110
+   -48₁₀ = 1101 0000
-----------------------
      6₁₀ = 0000 0110
```

```
     44₁₀ = 0010 1100
+   -48₁₀ = 1101 0000
-----------------------
     -4₁₀ = 1111 1100
```

```
    -44₁₀ = 1101 0100
+   -48₁₀ = 1101 0000
-----------------------
    -92₁₀ = 1010 0100
```

Two's Complement Overflow

- An overflow occurs if adding two positive numbers yields a negative result or if adding two negative numbers yields a positive result.
- Adding a positive and a negative number never causes an overflow.
- Carry out of the most significant bit does not indicate an overflow.
- An overflow occurs when the carry into the most significant bit differs from the carry out of the most significant bit.

Two's Complement Overflow Examples

$$\begin{array}{rrcl} & 54_{10} & = & 0011\ 0110 \\ + & 108_{10} & = & 0110\ 1100 \\ \hline & 162_{10} & \neq & 1010\ 0010 \end{array}$$

$$\begin{array}{rrcl} & -103_{10} & = & 1001\ 1001 \\ + & -48_{10} & = & 1101\ 0000 \\ \hline & -151_{10} & \neq & 0110\ 1001 \end{array}$$

Two's Complement Sign Extension

Decimal	8-bit	16-bit
+5	0000 0101	0000 0000 0000 0101
-5	1111 1011	1111 1111 1111 1011

- Why does sign extension work?

$-x$ is represented as $2^8 - x$ in 8-bit

$-x$ is represented as $2^{16} - x$ in 16-bit

$2^8 - x + ??? = 2^{16} - x$

$??? = 2^{16} - 2^8$

```
   1 0000 0000 0000 0000 = 65536
-              1 0000 0000 =   256
_________________________________
     1111 1111 0000 0000 = 65280
```

MODULO ARITHMETIC

Is Two's Complement "Magic"?

- Why does adding positive and negative numbers work?
- Why do we add 1 to the one's complement to negate?

- Answer: Because modulo arithmetic works.

Modulo Arithmetic

- Definition: Let a and b be integers and let m be a positive integer. We say that $a \equiv b \pmod{m}$ if the remainder of a divided by m is equal to the remainder of b divided by m.

- In the C programming language, $a \equiv b \pmod{m}$ would be written

```
a % m == b % m
```

- We use the theorem:

  If $a \equiv b \pmod{m}$ and $c \equiv d \pmod{m}$
  then $a + c \equiv b + d \pmod{m}$.

A Theorem of Modulo Arithmetic

Thm: If $a \equiv b \pmod{m}$ and $c \equiv d \pmod{m}$ then $a + c \equiv b + d \pmod{m}$.

Example: Let $m = 8$, $a = 3$, $b = 27$, $c = 2$ and $d = 18$.

$3 \equiv 27 \pmod{8}$ and $2 \equiv 18 \pmod{8}$.

$5 \equiv 45 \pmod{8}$.

Proof: Write $a = q_a m + r_a$, $b = q_b m + r_b$, $c = q_c m + r_c$ and $d = q_d m + r_d$, where r_a, r_b, r_c and r_d are between 0 and $m - 1$. Then,

$$a + c = (q_a + q_c)m + r_a + r_c$$

$$b + d = (q_b + q_d)m + r_b + r_d = (q_b + q_d)m + r_a + r_c.$$

Thus, $a + c \equiv r_a + r_c \equiv b + d \pmod{m}$.

Consider Numbers Modulo 256

$$\begin{array}{rcrcrcrcr}
0000\ 0000_2 & = & 0 & \equiv & -256 & \equiv & 256 & \equiv & 512 \\
0000\ 0001_2 & = & 1 & \equiv & -255 & \equiv & 257 & \equiv & 513 \\
0000\ 0010_2 & = & 2 & \equiv & -254 & \equiv & 258 & \equiv & 514 \\
\vdots & & & & & & & & \\
0000\ 1111_2 & = & 15 & \equiv & -241 & \equiv & 271 & \equiv & 527 \\
\vdots & & & & & & & & \\
0111\ 1111_2 & = & 127 & \equiv & -129 & \equiv & 383 & \equiv & 639 \\
1000\ 0000_2 & = & 128 & \equiv & -128 & \equiv & 384 & \equiv & 640 \\
\vdots & & & & & & & & \\
1000\ 1111_2 & = & 143 & \equiv & -113 & \equiv & 399 & \equiv & 655 \\
\vdots & & & & & & & & \\
1111\ 0011_2 & = & 243 & \equiv & -13 & \equiv & 499 & \equiv & 755 \\
\vdots & & & & & & & & \\
1111\ 1111_2 & = & 255 & \equiv & -1 & \equiv & 511 & \equiv & 767
\end{array}$$

If $0000\ 0000_2$ thru $0111\ 1111_2$ represents 0 thru 127 and $1000\ 0000_2$ thru $1111\ 1111_2$ represents -128 thru -1, then the most significant bit can be used to determine the sign.

Some Answers

- In 8-bit two's complement, we use addition modulo $2^8 = 256$, so adding 256 or subtracting 256 is equivalent to adding 0 or subtracting 0.

- To negate a number x, $0 \le x \le 128$:

$$-x = 0 - x \equiv 256 - x = (255 - x) + 1 = (\texttt{1111 1111}_2 - x) + 1$$

Note that $\texttt{1111 1111}_2 - x$ is the one's complement of x.

- Now we can just add positive and negative numbers. For example:

$$3 + (-5) \equiv 3 + (256 - 5) = 3 + 251 = 254 \equiv 254 - 256 = -2.$$

or two negative numbers (as long as there's no overflow):

$$(-3) + (-5) \equiv (256 - 3) + (256 - 5) = 504 \equiv 504 - 512 = -8.$$

FLOATING POINT NUMBERS

2.5 Floating-Point Representation

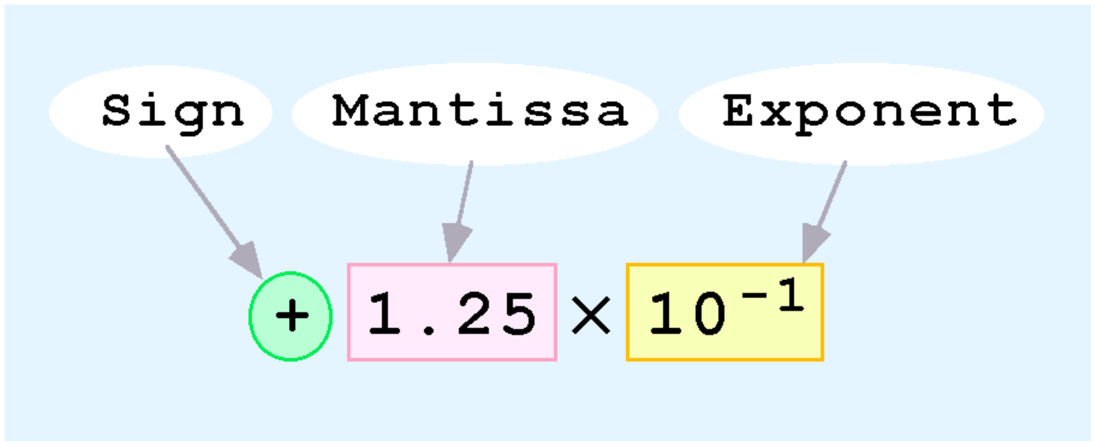
- Floating-point numbers allow an arbitrary number of decimal places to the right of the decimal point.
  - For example: $0.5 \times 0.25 = 0.125$
- They are often expressed in scientific notation.
  - For example:

    $0.125 = 1.25 \times 10^{-1}$

    $5,000,000 = 5.0 \times 10^{6}$

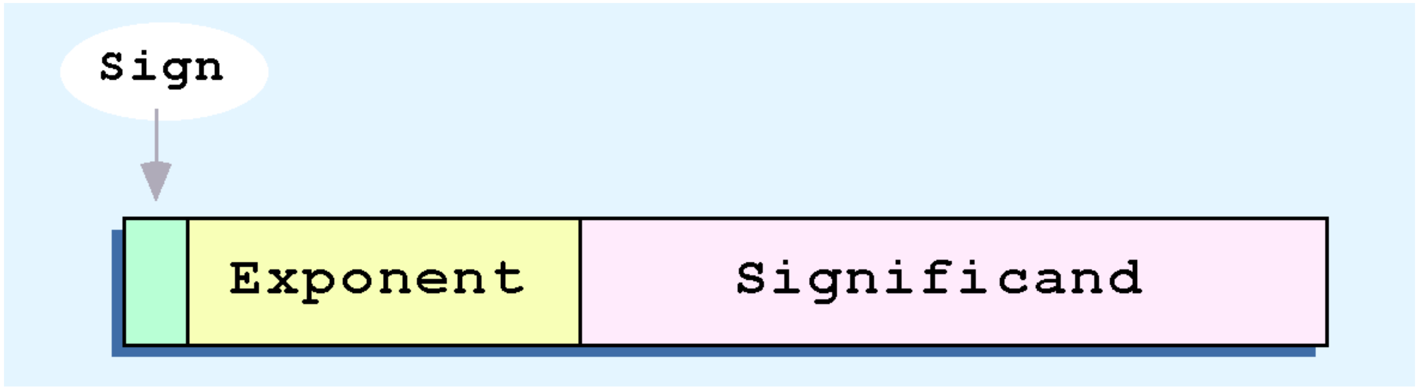
2.5 Floating-Point Representation

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



2.5 Floating-Point Representation

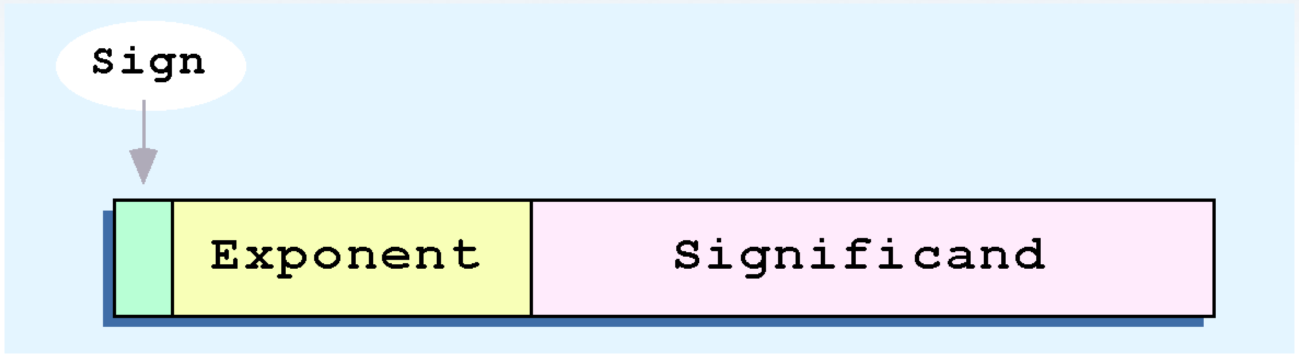
- Computer representation of a floating-point number consists of three fixed-size fields:



- This is the standard arrangement of these fields.

Note: Although "significand" and "mantissa" do not technically mean the same thing, many people use these terms interchangeably. We use the term "significand" to refer to the fractional part of a floating point number.

2.5 Floating-Point Representation



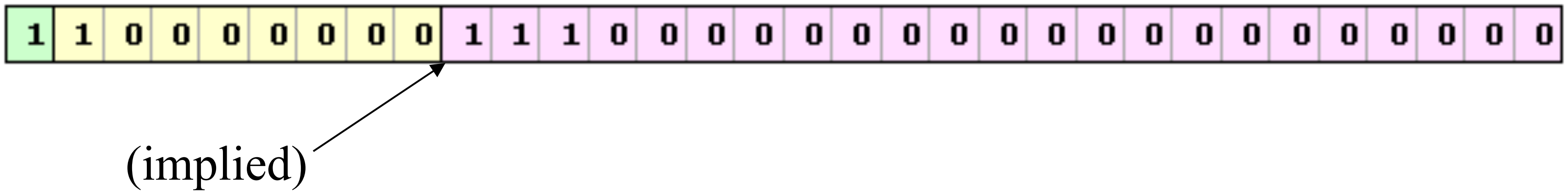
- The one-bit sign field is the sign of the stored value.
- The size of the exponent field determines the range of values that can be represented.
- The size of the significand determines the precision of the representation.

IEEE-754 32-bit Floating Point Format

- sign bit, 8-bit exponent, 23-bit mantissa
- normalized as 1.xxxxx
- leading 1 is hidden
- 8-bit exponent in excess 127 format
  - NOT excess 128
  - `0000 0000` and `1111 1111` are reserved
- +0 and -0 is zero exponent and zero mantissa
- `1111 1111` exponent and zero mantissa is infinity

2.5 Floating-Point Representation

- Example: Express -3.75 as a floating point number using IEEE single precision.
- First, let's normalize according to IEEE rules:
  - 3.75 = -11.11_2 = -1.111×2^1
  - The bias is 127, so we add 127 + 1 = 128 (this is our exponent)



  - Since we have an implied 1 in the significand, this equates to

    $-(1).111_2 \times 2^{(128-127)} = -1.111_2 \times 2^1 = -11.11_2 = -3.75.$

2.5 Floating-Point Representation

- Using the IEEE-754 single precision floating point standard:
  - An exponent of 255 indicates a special value.
    - If the significand is zero, the value is ± infinity.
    - If the significand is nonzero, the value is NaN, "not a number," often used to flag an error condition.
- Using the double precision standard:
  - The "special" exponent value for a double precision number is 2047, instead of the 255 used by the single precision standard.

CHARACTERS & STRINGS

2.6 Character Codes

- Calculations aren’t useful until their results can be displayed in a manner that is meaningful to people.
- We also need to store the results of calculations, and provide a means for data input.
- Thus, human-understandable characters must be converted to computer-understandable bit patterns using some sort of character encoding scheme.

2.6 Character Codes

- As computers have evolved, character codes have evolved.
- Larger computer memories and storage devices permit richer character codes.
- The earliest computer coding systems used six bits.
- Binary-coded decimal (BCD) was one of these early codes. It was used by IBM mainframes in the 1950s and 1960s.

2.6 Character Codes

- In 1964, BCD was extended to an 8-bit code, Extended Binary-Coded Decimal Interchange Code (EBCDIC).
- EBCDIC was one of the first widely-used computer codes that supported upper *and* lowercase alphabetic characters, in addition to special characters, such as punctuation and control characters.
- EBCDIC and BCD are still in use by IBM mainframes today.

EBCDIC Character Code

- EBCDIC is an 8-bit code.

00	NUL	20	DS	40	SP	60	–	80		A0		C0	{	E0	\
01	SOH	21	SOS	41		61	/	81	a	A1	~	C1	A	E1	
02	STX	22	FS	42		62		82	b	A2	s	C2	B	E2	S
03	ETX	23		43		63		83	c	A3	t	C3	C	E3	T
04	PF	24	BYP	44		64		84	d	A4	u	C4	D	E4	U
05	HT	25	LF	45		65		85	e	A5	v	C5	E	E5	V
06	LC	26	ETB	46		66		86	f	A6	w	C6	F	E6	W
07	DEL	27	ESC	47		67		87	g	A7	x	C7	G	E7	X
08		28		48		68		88	h	A8	y	C8	H	E8	Y
09		29		49		69		89	i	A9	z	C9	I	E9	Z
0A	SMM	2A	SM	4A	¢	6A	‘	8A		AA		CA		EA	
0B	VT	2B	CU2	4B		6B	,	8B		AB		CB		EB	
0C	FF	2C		4C	<	6C	%	8C		AC		CC		EC	
0D	CR	2D	ENQ	4D	(	6D	_	8D		AD		CD		ED	
0E	SO	2E	ACK	4E	+	6E	>	8E		AE		CE		EE	
0F	SI	2F	BEL	4F	|	6F	?	8F		AF		CF		EF	
10	DLE	30		50	&	70		90		B0		D0	}	F0	0
11	DC1	31		51		71		91	j	B1		D1	J	F1	1
12	DC2	32	SYN	52		72		92	k	B2		D2	K	F2	2
13	TM	33		53		73		93	l	B3		D3	L	F3	3
14	RES	34	PN	54		74		94	m	B4		D4	M	F4	4
15	NL	35	RS	55		75		95	n	B5		D5	N	F5	5
16	BS	36	UC	56		76		96	o	B6		D6	O	F6	6
17	IL	37	EOT	57		77		97	p	B7		D7	P	F7	7
18	CAN	38		58		78		98	q	B8		D8	Q	F8	8
19	EM	39		59		79		99	r	B9		D9	R	F9	9
1A	CC	3A		5A	!	7A	:	9A		BA		DA		FA	|
1B	CU1	3B	CU3	5B	$	7B	#	9B		BB		DB		FB	
1C	IFS	3C	DC4	5C	·	7C	@	9C		BC		DC		FC	
1D	IGS	3D	NAK	5D	)	7D	'	9D		BD		DD		FD	
1E	IRS	3E		5E	;	7E	=	9E		BE		DE		FE	
1F	IUS	3F	SUB	5F	¬	7F	"	9F		BF		DF		FF	

STX	Start of text	RS	Reader Stop	DC
DLE	Data Link Escape	PF	Punch Off	DC
BS	Backspace	DS	Digit Select	DC
ACK	Acknowledge	PN	Punch On	CU
SOH	Start of Heading	SM	Set Mode	CU
ENQ	Enquiry	LC	Lower Case	CU
ESC	Escape	CC	Cursor Control	SY
BYP	Bypass	CR	Carriage Return	IF
CAN	Cancel	EM	End of Medium	EC
RES	Restore	FF	Form Feed	ET
SI	Shift In	TM	Tape Mark	NA
SO	Shift Out	UC	Upper Case	SM
DEL	Delete	FS	Field Separator	SC
SUB	Substitute	HT	Horizontal Tab	IG
NL	New Line	VT	Vertical Tab	IR
LF	Line Feed	UC	Upper Case	IU

2.6 Character Codes

- Other computer manufacturers chose the 7-bit ASCII (American Standard Code for Information Interchange) as a replacement for 6-bit codes.
- While BCD and EBCDIC were based upon punched card codes, ASCII was based upon telecommunications (Telex) codes.
- Until recently, ASCII was the dominant character code outside the IBM mainframe world.

ASCII Character Code

- **ASCII is a 7-bit code, commonly stored in 8-bit bytes.**
- **“A” is at 41_{16}. To convert upper case letters to lower case letters, add 20_{16}. Thus “a” is at $41_{16} + 20_{16} = 61_{16}$.**
- **The character “5” at position 35_{16} is different than the number 5. To convert character-numbers into number-numbers, subtract 30_{16}: $35_{16} - 30_{16} = 5$.**

00	NUL	10	DLE	20	SP	30	0	40	@	50	P	60	`	70	p
01	SOH	11	DC1	21	!	31	1	41	A	51	Q	61	a	71	q
02	STX	12	DC2	22	"	32	2	42	B	52	R	62	b	72	r
03	ETX	13	DC3	23	#	33	3	43	C	53	S	63	c	73	s
04	EOT	14	DC4	24	$	34	4	44	D	54	T	64	d	74	t
05	ENQ	15	NAK	25	%	35	5	45	E	55	U	65	e	75	u
06	ACK	16	SYN	26	&	36	6	46	F	56	V	66	f	76	v
07	BEL	17	ETB	27	'	37	7	47	G	57	W	67	g	77	w
08	BS	18	CAN	28	(	38	8	48	H	58	X	68	h	78	x
09	HT	19	EM	29	)	39	9	49	I	59	Y	69	i	79	y
0A	LF	1A	SUB	2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
0B	VT	1B	ESC	2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
0C	FF	1C	FS	2C	´	3C	<	4C	L	5C	\	6C	l	7C	|
0D	CR	1D	GS	2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
0E	SO	1E	RS	2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
0F	SI	1F	US	2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

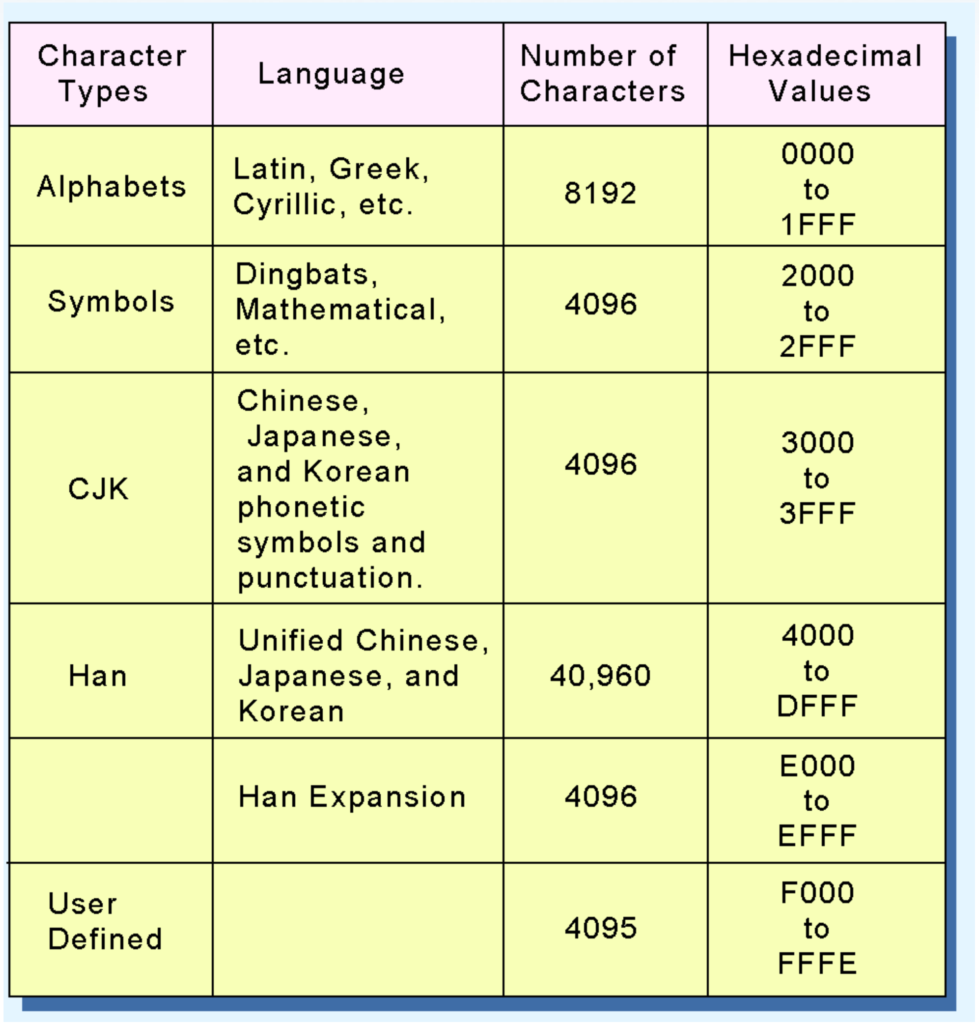
NUL	Null	FF	Form feed	CAN	Cancel
SOH	Start of heading	CR	Carriage return	EM	End of medium
STX	Start of text	SO	Shift out	SUB	Substitute
ETX	End of text	SI	Shift in	ESC	Escape
EOT	End of transmission	DLE	Data link escape	FS	File separator
ENQ	Enquiry	DC1	Device control 1	GS	Group separator
ACK	Acknowledge	DC2	Device control 2	RS	Record separator
BEL	Bell	DC3	Device control 3	US	Unit separator
BS	Backspace	DC4	Device control 4	SP	Space
HT	Horizontal tab	NAK	Negative acknowledge	DEL	Delete
LF	Line feed	SYN	Synchronous idle		
VT	Vertical tab	ETB	End of transmission block		

2.6 Character Codes

- Many of today's systems embrace Unicode, a 16-bit system 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.
- The Unicode codespace is divided into six parts. The first part is for Western alphabet codes, including English, Greek, and Russian.

2.6 Character Codes

- The Unicode codes-pace allocation is shown at the right.
- The lowest-numbered Unicode characters comprise the ASCII code.
- The highest provide for user-defined codes.



Character Types	Language	Number of Characters	Hexadecimal Values
Alphabets	Latin, Greek, Cyrillic, etc.	8192	0000 to 1FFF
Symbols	Dingbats, Mathematical, etc.	4096	2000 to 2FFF
CJK	Chinese, Japanese, and Korean phonetic symbols and punctuation.	4096	3000 to 3FFF
Han	Unified Chinese, Japanese, and Korean	40,960	4000 to DFFF
	Han Expansion	4096	E000 to EFFF
User Defined		4095	F000 to FFFE

Unicode Character Code

- **Unicode is a 16-bit code.**

0000	NUL	0020	SP	0040	@	0060	`	0080	Ctrl	00A0	NBS	00C0	À	00E0	à
0001	SOH	0021	!	0041	A	0061	a	0081	Ctrl	00A1	¡	00C1	Á	00E1	á
0002	STX	0022	"	0042	B	0062	b	0082	Ctrl	00A2	¢	00C2	Â	00E2	â
0003	ETX	0023	#	0043	C	0063	c	0083	Ctrl	00A3	£	00C3	Ã	00E3	ã
0004	EOT	0024	$	0044	D	0064	d	0084	Ctrl	00A4	¤	00C4	Ä	00E4	ä
0005	ENQ	0025	%	0045	E	0065	e	0085	Ctrl	00A5	¥	00C5	Å	00E5	å
0006	ACK	0026	&	0046	F	0066	f	0086	Ctrl	00A6	¦	00C6	Æ	00E6	æ
0007	BEL	0027	'	0047	G	0067	g	0087	Ctrl	00A7	§	00C7	Ç	00E7	ç
0008	BS	0028	(	0048	H	0068	h	0088	Ctrl	00A8	¨	00C8	È	00E8	è
0009	HT	0029	)	0049	I	0069	i	0089	Ctrl	00A9	©	00C9	É	00E9	é
000A	LF	002A	*	004A	J	006A	j	008A	Ctrl	00AA	ª	00CA	Ê	00EA	ê
000B	VT	002B	+	004B	K	006B	k	008B	Ctrl	00AB	«	00CB	Ë	00EB	ë
000C	FF	002C	´	004C	L	006C	l	008C	Ctrl	00AC	¬	00CC	Ì	00EC	ì
000D	CR	002D	-	004D	M	006D	m	008D	Ctrl	00AD	–	00CD	Í	00ED	í
000E	SO	002E	.	004E	N	006E	n	008E	Ctrl	00AE	®	00CE	Î	00EE	î
000F	SI	002F	/	004F	O	006F	o	008F	Ctrl	00AF	¯	00CF	Ï	00EF	ï
0010	DLE	0030	0	0050	P	0070	p	0090	Ctrl	00B0	°	00D0	Đ	00F0	¶́
0011	DC1	0031	1	0051	Q	0071	q	0091	Ctrl	00B1	±	00D1	Ñ	00F1	ñ
0012	DC2	0032	2	0052	R	0072	r	0092	Ctrl	00B2	^{2}	00D2	Ò	00F2	ò
0013	DC3	0033	3	0053	S	0073	s	0093	Ctrl	00B3	^{3}	00D3	Ó	00F3	ó
0014	DC4	0034	4	0054	T	0074	t	0094	Ctrl	00B4	´	00D4	Ô	00F4	ô
0015	NAK	0035	5	0055	U	0075	u	0095	Ctrl	00B5	µ	00D5	Õ	00F5	õ
0016	SYN	0036	6	0056	V	0076	v	0096	Ctrl	00B6	¶	00D6	Ö	00F6	ö
0017	ETB	0037	7	0057	W	0077	w	0097	Ctrl	00B7	·	00D7	×	00F7	÷
0018	CAN	0038	8	0058	X	0078	x	0098	Ctrl	00B8	¸	00D8	Ø	00F8	ø
0019	EM	0039	9	0059	Y	0079	y	0099	Ctrl	00B9	^{1}	00D9	Ù	00F9	ù
001A	SUB	003A	:	005A	Z	007A	z	009A	Ctrl	00BA	º	00DA	Ú	00FA	ú
001B	ESC	003B	;	005B	[	007B	{	009B	Ctrl	00BB	»	00DB	Û	00FB	û
001C	FS	003C	<	005C	\	007C	|	009C	Ctrl	00BC	1/4	00DC	Ü	00FC	ü
001D	GS	003D	=	005D	]	007D	}	009D	Ctrl	00BD	1/2	00DD	Ý	00FD	Ꝥ
001E	RS	003E	>	005E	^	007E	~	009E	Ctrl	00BE	3/4	00DE	ý	00FE	þ
001F	US	003F	?	005F	_	007F	DEL	009F	Ctrl	00BF	¿	00DF	§	00FF	ÿ

NUL	Null	SOH	Start of heading	CAN	Cancel	SP	Space
STX	Start of text	EOT	End of transmission	EM	End of medium	DEL	Delete
ETX	End of text	DC1	Device control 1	SUB	Substitute	Ctrl	Control
ENQ	Enquiry	DC2	Device control 2	ESC	Escape	FF	Form feed
ACK	Acknowledge	DC3	Device control 3	FS	File separator	CR	Carriage return
BEL	Bell	DC4	Device control 4	GS	Group separator	SO	Shift out
BS	Backspace	NAK	Negative acknowledge	RS	Record separator	SI	Shift in
HT	Horizontal tab	NBS	Non-breaking space	US	Unit separator	DLE	Data link escape
LF	Line feed	ETB	End of transmission block	SYN	Synchronous idle	VT	Vertical tab

MEMORY HAS NO TYPE!

A single byte in memory might be

- a character
- an unsigned number
- a signed number
- part of a multi-byte integer in little endian
- part of a multi-byte integer in big endian
- part of a multi-byte floating point number
- ...

NEXT TIME

- Basic Intel i-386 architecture
- “Hello World” in Linux assembly
- Addressing modes