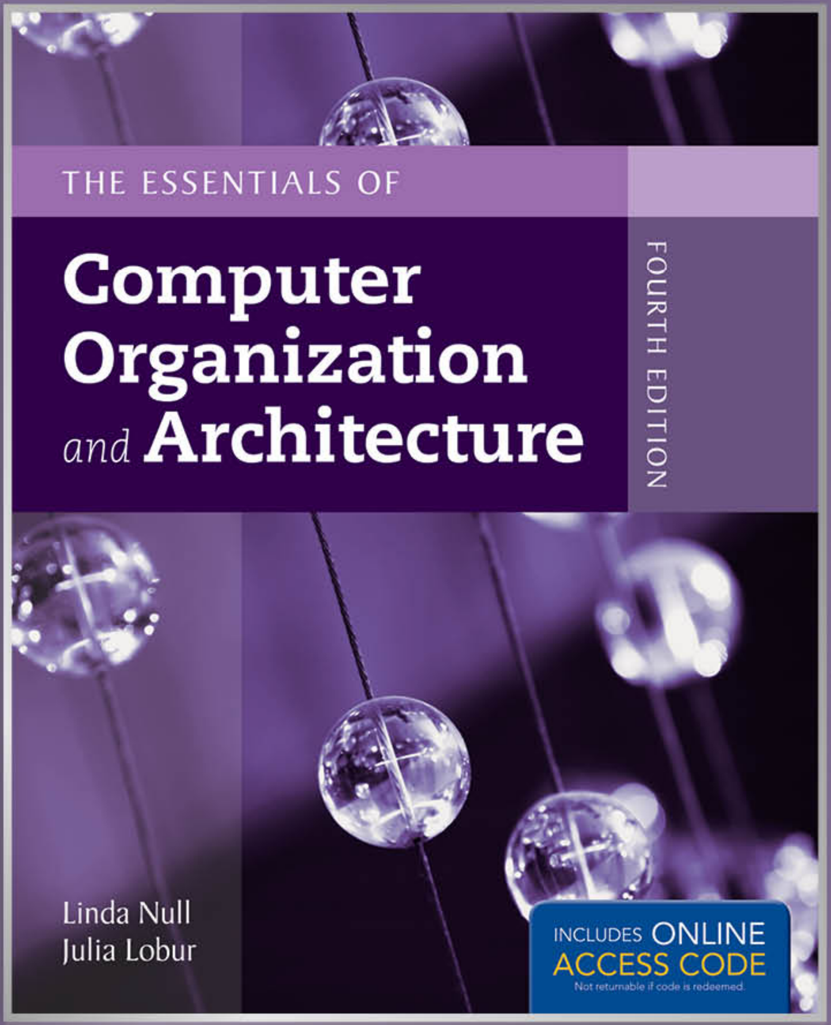
Chapter 2

Data Representation in Computer Systems


THE ESSENTIALS OF
Computer Organization and Architecture
FOURTH EDITION
Linda Null
Julia Lobur
INCLUDES ONLINE ACCESS CODE
Not returnable if code is redeemed.

2.5 Floating-Point Representation

- The signed magnitude, one’s complement, and two’s complement representation that we have just presented deal with signed integer values only.
- Without modification, these formats are not useful in scientific or business applications that deal with real number values.
- Floating-point representation solves this problem.

2.5 Floating-Point Representation

- If we are clever programmers, we can perform floating-point calculations using any integer format.
- This is called *floating-point emulation*, because floating point values aren't stored as such; we just create programs that make it seem as if floating-point values are being used.
- Most of today's computers are equipped with specialized hardware that performs floating-point arithmetic with no special programming required.

2.5 Floating-Point Representation

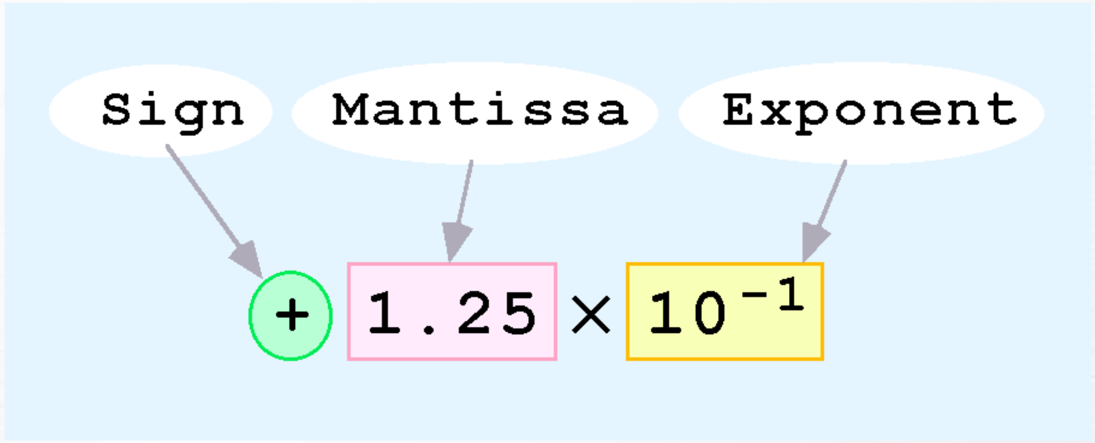
- Floating-point numbers allow an arbitrary number of decimal places to the right of the decimal point.
  - For example: $0.5 \times 0.25 = 0.125$
- They are often expressed in scientific notation.
  - For example:

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

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

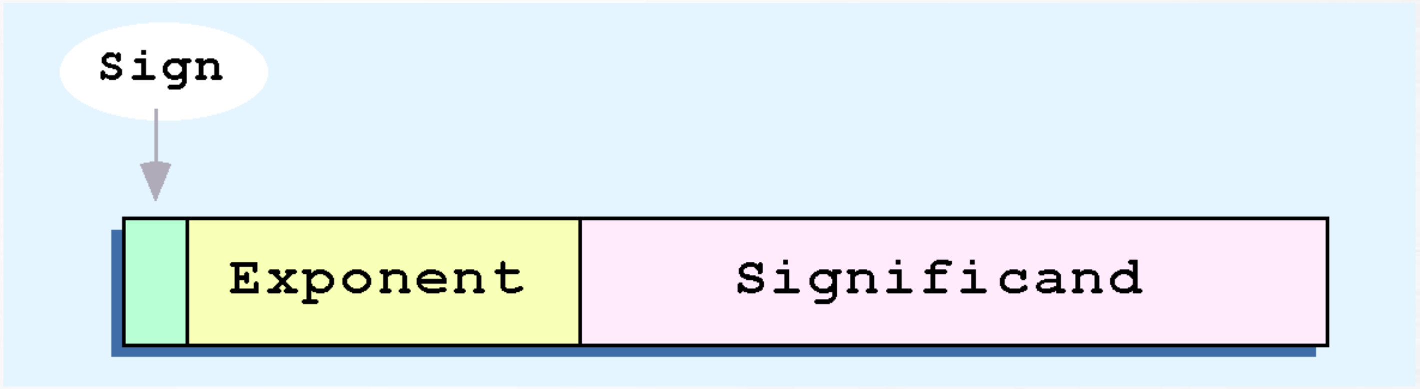
2.5 Floating-Point Representation

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



2.5 Floating-Point Representation

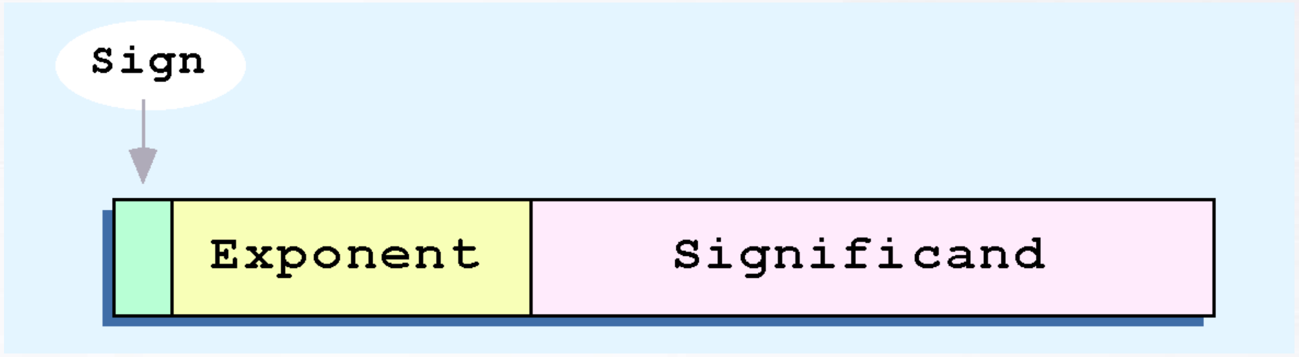
- Computer representation of a floating-point number consists of three fixed-size fields:



- This is the standard arrangement of these fields.

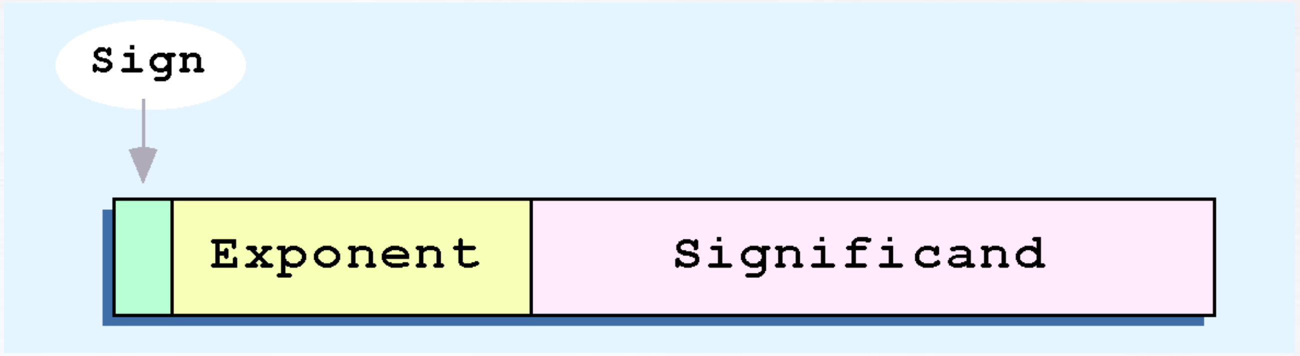
Note: Although "significand" and "mantissa" do not technically mean the same thing, many people use these terms interchangeably. We use the term "significand" to refer to the fractional part of a floating point number.

2.5 Floating-Point Representation



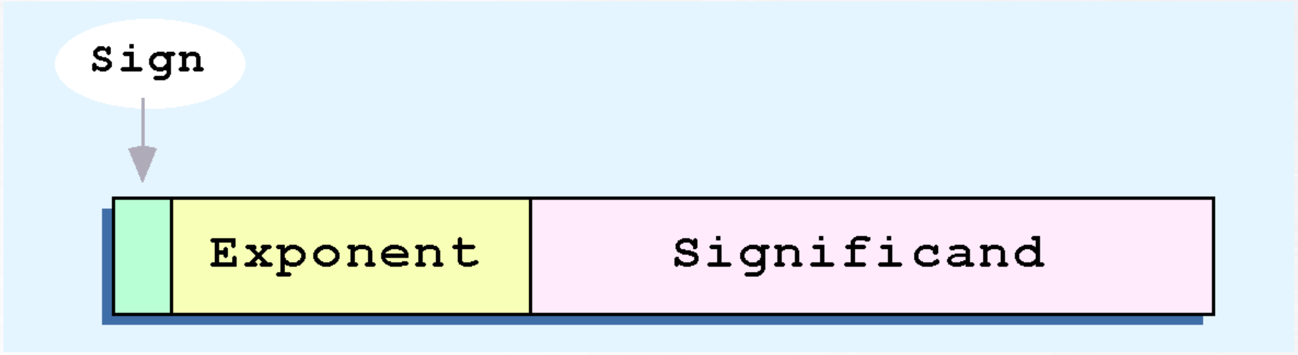
- The one-bit sign field is the sign of the stored value.
- The size of the exponent field determines the range of values that can be represented.
- The size of the significand determines the precision of the representation.

2.5 Floating-Point Representation



- We introduce a hypothetical "Simple Model" to explain the concepts
- In this model:
  - A floating-point number is 14 bits in length
  - The exponent field is 5 bits
  - The significand field is 8 bits

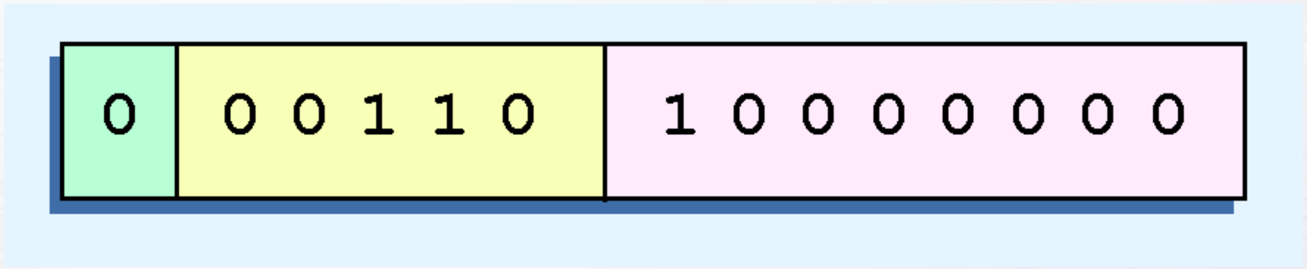
2.5 Floating-Point Representation



- The significand is always preceded by an implied binary point.
- Thus, the significand always contains a fractional binary value.
- The exponent indicates the power of 2 by which the significand is multiplied.

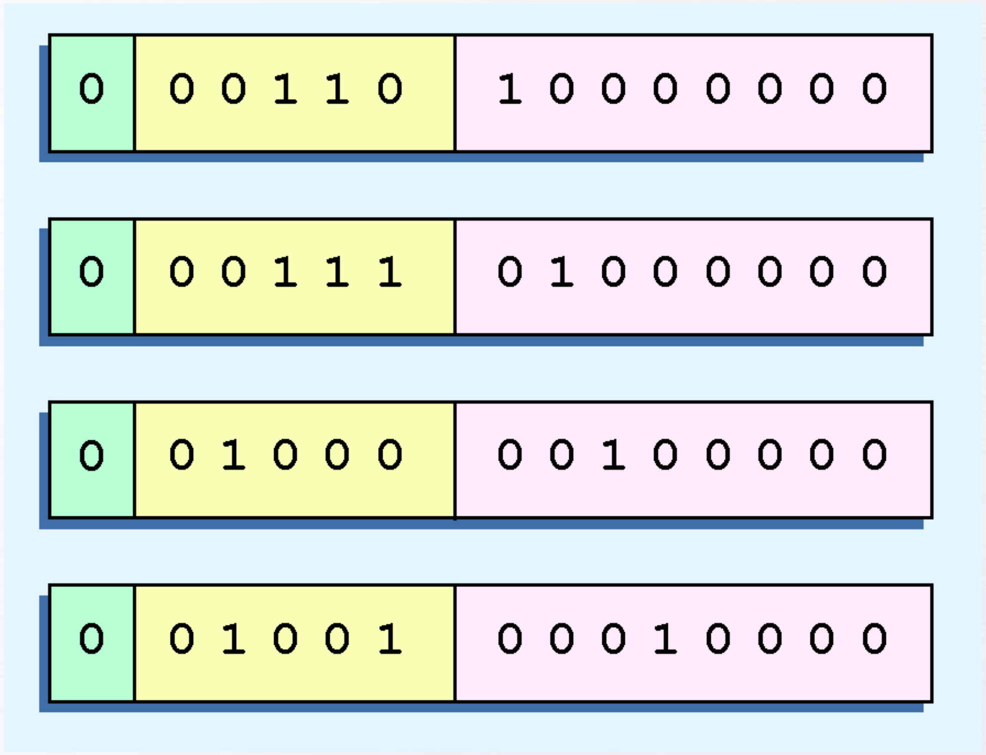
2.5 Floating-Point Representation

- Example:
  - Express 32_{10} in the simplified 14-bit floating-point model.
- We know that 32 is 2^5. So in (binary) scientific notation $32 = 1.0 \times 2^5 = 0.1 \times 2^6$.
  - In a moment, we'll explain why we prefer the second notation versus the first.
- Using this information, we put 110 (= 6_{10}) in the exponent field and 1 in the significand as shown.

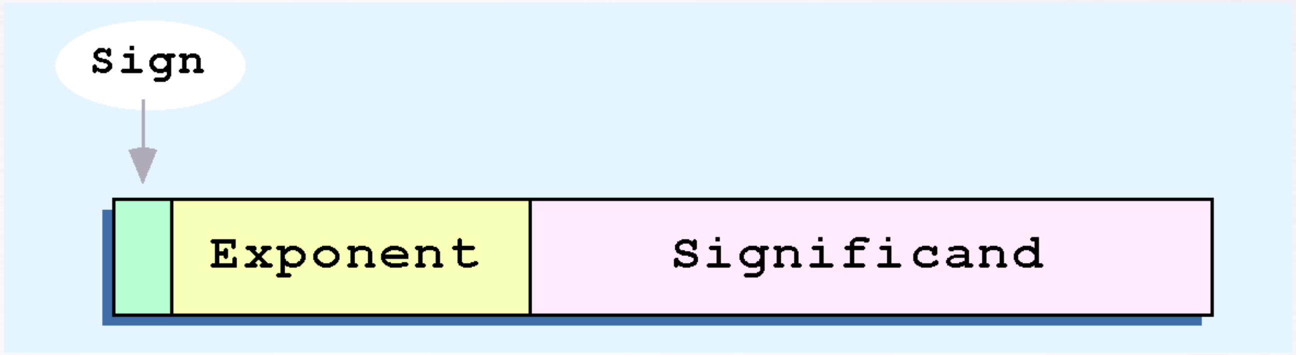


2.5 Floating-Point Representation

- The illustrations shown at the right are *all* equivalent representations for 32 using our simplified model.
- Not only do these synonymous representations waste space, but they can also cause confusion.



2.5 Floating-Point Representation



- Another problem with our system is that we have made no allowances for negative exponents. We have no way to express 0.5 (=2 $^{-1}$)! (Notice that there is no sign in the exponent field.)

All of these problems can be fixed with no changes to our basic model.

2.5 Floating-Point Representation

- To resolve the problem of synonymous forms, we establish a rule that the first digit of the significand must be 1, with no ones to the left of the radix point.
- This process, called *normalization*, results in a unique pattern for each floating-point number.
  - In our simple model, all significands must have the form 0.1xxxxxxxx
  - For example, $4.5 = 100.1 \times 2^0 = 1.001 \times 2^2 = 0.1001 \times 2^3$. The last expression is correctly

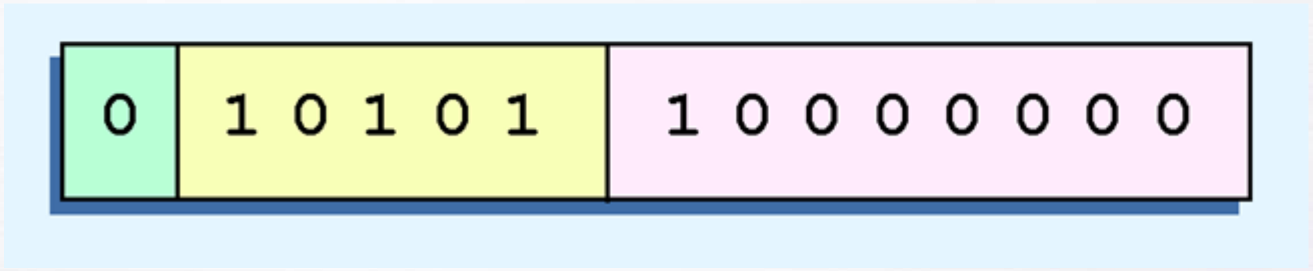
In our simple instructional model, we use no implied bits.

2.5 Floating-Point Representation

- To provide for negative exponents, we will use a *biased exponent*.
  - In our case, we have a 5-bit exponent.
  - $2^{5-1} - 1 = 2^4 - 1 = 15$
  - Thus will use 15 for our bias: our exponent will use *excess-15* representation.
- In our model, exponent values less than 15 are negative, representing fractional numbers.

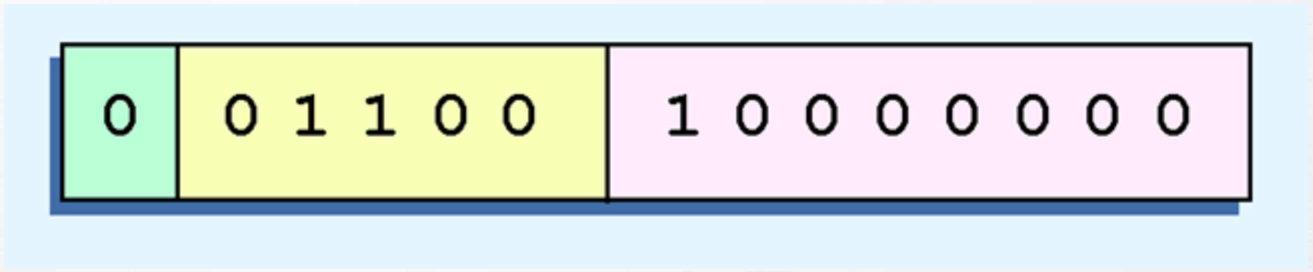
2.5 Floating-Point Representation

- Example:
  - Express 32_{10} in the revised 14-bit floating-point model.
- We know that $32 = 1.0 \times 2^5 = 0.1 \times 2^6$.
- To use our excess 15 biased exponent, we add 15 to 6, giving 21_{10} ($=10101_2$).
- So we have:



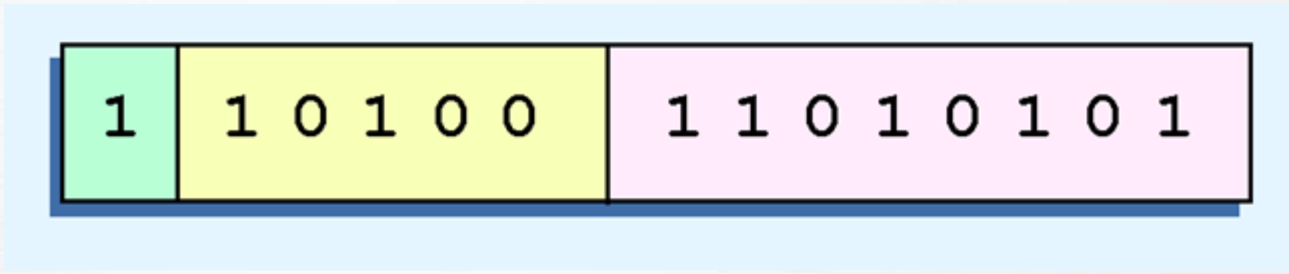
2.5 Floating-Point Representation

- Example:
  - Express 0.0625_{10} in the revised 14-bit floating-point model.
- We know that 0.0625 is 2^{-4}. So in (binary) scientific notation $0.0625 = 1.0 \times 2^{-4} = 0.1 \times 2^{-3}$.
- To use our excess 15 biased exponent, we add 15 to -3, giving 12_{10} ($=01100_2$).



2.5 Floating-Point Representation

- Example:
  - Express -26.625_{10} in the revised 14-bit floating-point model.
- We find $26.625_{10} = 11010.101_2$. Normalizing, we have: $26.625_{10} = 0.11010101 \times 2^5$.
- To use our excess 15 biased exponent, we add 15 to 5, giving 20_{10} ($=10100_2$). We also need a 1 in the sign bit.



2.5 Floating-Point Representation

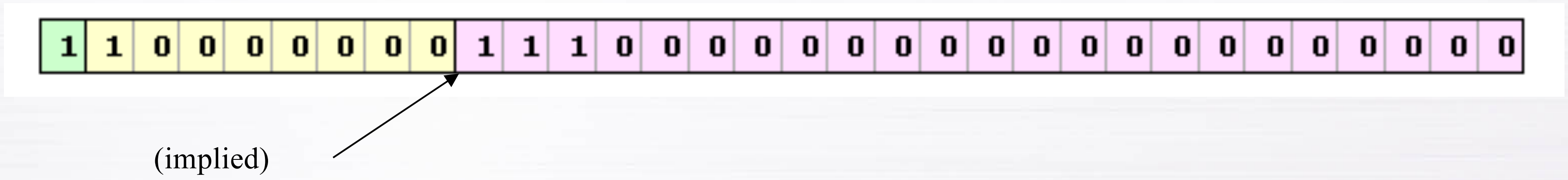
- The IEEE has established a standard for floating-point numbers
- The IEEE-754 *single precision* floating point standard uses an 8-bit exponent (with a bias of 127) and a 23-bit significand.
- The IEEE-754 *double precision* standard uses an 11-bit exponent (with a bias of 1023) and a 52-bit significand.

2.5 Floating-Point Representation

- In both the IEEE single-precision and double-precision floating-point standard, the significant has an implied 1 to the LEFT of the radix point.
  - The format for a significand using the IEEE format is: 1.xxx…
  - For example, 4.5 = .1001 x 2^3 in IEEE format is 4.5 = 1.001 x 2^2. The 1 is implied, which means is does not need to be listed in the significand (the significand would include only 001).

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.

2.5 Floating-Point Representation

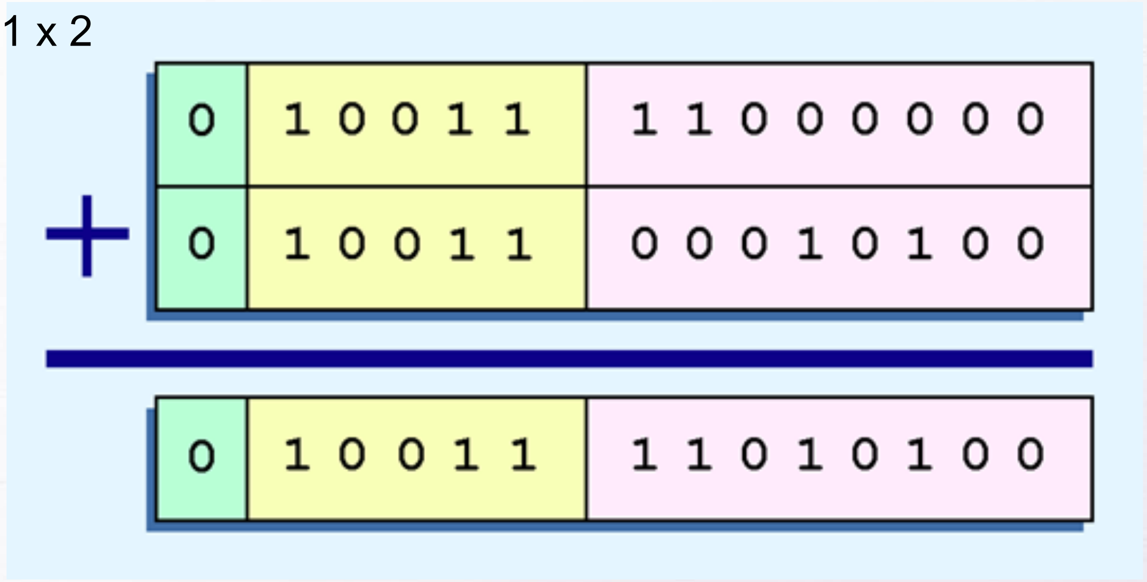
- Both the 14-bit model that we have presented and the IEEE-754 floating point standard allow two representations for zero.
  - Zero is indicated by all zeros in the exponent and the significand, but the sign bit can be either 0 or 1.
- This is why programmers should avoid testing a floating-point value for equality to zero.
  - Negative zero does not equal positive zero.

2.5 Floating-Point Representation

- Floating-point addition and subtraction are done using methods analogous to how we perform calculations using pencil and paper.
- The first thing that we do is express both operands in the same exponential power, then add the numbers, preserving the exponent in the sum.
- If the exponent requires adjustment, we do so at the end of the calculation.

2.5 Floating-Point Representation

- Example:
  - Find the sum of 12_{10} and 1.25_{10} using the 14-bit "simple" floating-point model.
- We find $12_{10} = 0.1100 \times 2^4$. And $1.25_{10} = 0.101 \times 2^1 = 0.000101 \times 2^4$.
- Thus, our sum is 0.110101×2^4.

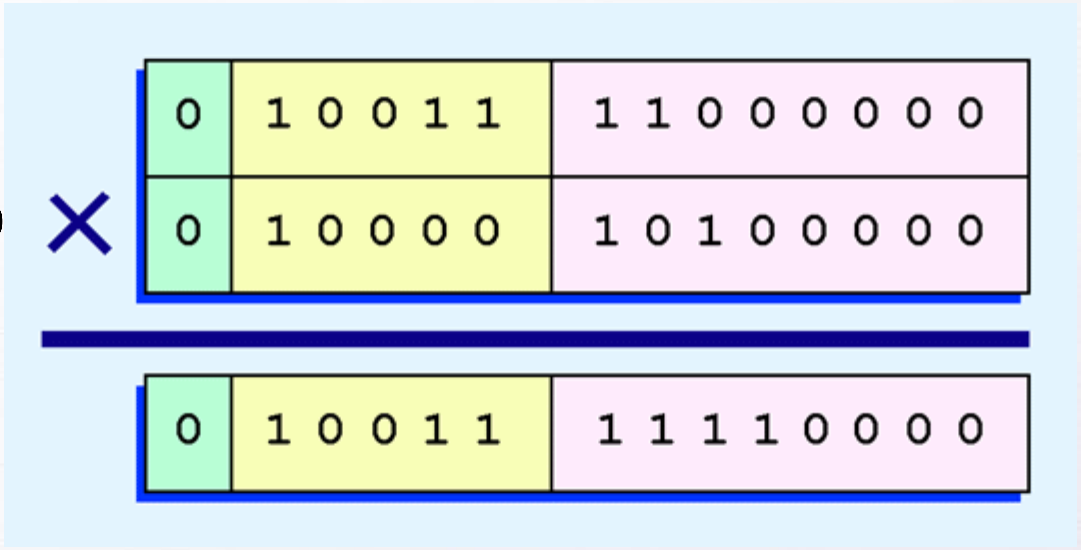


2.5 Floating-Point Representation

- Floating-point multiplication is also carried out in a manner akin to how we perform multiplication using pencil and paper.
- We multiply the two operands and add their exponents.
- If the exponent requires adjustment, we do so at the end of the calculation.

2.5 Floating-Point Representation

- Example:
  - Find the product of 12_{10} and 1.25_{10} using the 14-bit floating-point model.
- We find $12_{10} = 0.1100 \times 2^{4}$. And $1.25_{10} = 0.101 \times 2^{1}$.
- Thus, our product is $0.0111100 \times 2^{5} = 0.1111 \times 2^{4}$.
- The normalized product requires an exponent of $19_{10} = 10011_2$.



2.5 Floating-Point Representation

- No matter how many bits we use in a floating-point representation, our model must be finite.
- The real number system is, of course, infinite, so our models can give nothing more than an approximation of a real value.
- 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.

2.5 Floating-Point Representation

- Our job becomes one of reducing error, or at least being aware of the possible magnitude of error in our calculations.
- We must also be aware that errors can compound through repetitive arithmetic operations.
- For example, our 14-bit model cannot exactly represent the decimal value 128.5. In binary, it is 9 bits wide:

  $10000000.1_2 = 128.5_{10}$

2.5 Floating-Point Representation

- When we try to express 128.5_{10} in our 14-bit model, we lose the low-order bit, giving a relative error of:

$$\frac{128.5 - 128}{128.5} \approx 0.39\%$$

- If we had a procedure that repetitively added 0.5 to 128.5, we would have an error of nearly 2% after only four iterations.

2.5 Floating-Point Representation

- Floating-point errors can be reduced when we use operands that are similar in magnitude.
- If we were repetitively adding 0.5 to 128.5, it would have been better to iteratively add 0.5 to itself and then add 128.5 to this sum.
- In this example, the error was caused by loss of the low-order bit.
- Loss of the high-order bit is more problematic.

2.5 Floating-Point Representation

- Floating-point overflow and underflow can cause programs to crash.
- Overflow occurs when there is no room to store the high-order bits resulting from a calculation.
- Underflow occurs when a value is too small to store, possibly resulting in division by zero.

Experienced programmers know that it's better for a program to crash than to have it produce incorrect, but plausible, results.

2.5 Floating-Point Representation

- When discussing floating-point numbers, it is important to understand the terms *range, precision,* and *accuracy*.
- The range of a numeric integer format is the difference between the largest and smallest values that can be expressed.
- Accuracy refers to how closely a numeric representation approximates a true value.
- The precision of a number indicates how much information we have about a value

2.5 Floating-Point Representation

- Most of the time, greater precision leads to better accuracy, but this is not always true.
  - For example, 3.1333 is a value of pi that is accurate to two digits, but has 5 digits of precision.
- There are other problems with floating point numbers.
- Because of truncated bits, you cannot always assume that a particular floating point operation is commutative or distributive.

2.5 Floating-Point Representation

- This means that we cannot assume:

(a + b) + c = a + (b + c) or
a*(b + c) = ab + ac

- Moreover, to test a floating point value for equality to some other number, it is best to declare a “nearness to x” epsilon value. For example, instead of checking to see if floating point x is equal to 2 as follows:

if x = 2 then …

it is better to use:

if (abs(x - 2) < epsilon) then ...

(assuming we have epsilon defined correctly!)

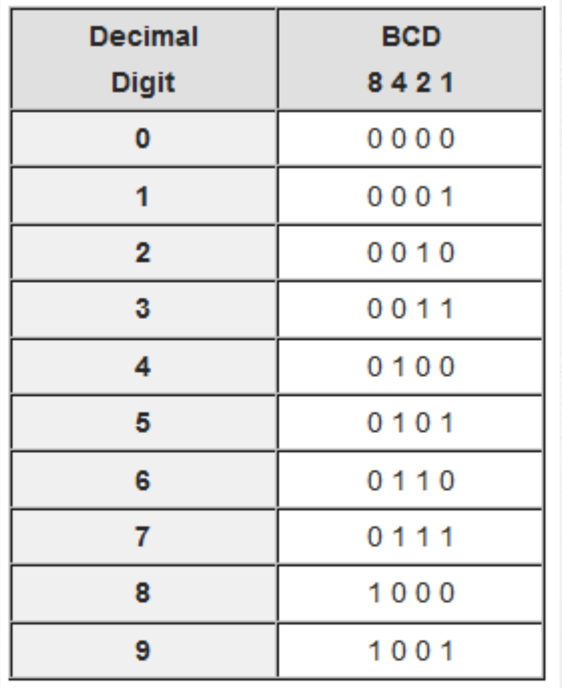
2.6 Character Codes

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

2.6.1 Binary-Coded Decimal

- For many applications decimal digits need to be exactly represented, e.g. financial applications.
  - What is the binary equivalent of R 5.10?
- For these cases, we need an encoding for individual decimal digits (a form of character coding).
- Binary-coded decimal (BCD) is common in electronics that use numerical data, such as alarm clocks and calculators.
- BCD encodes each digit of a decimal number into a 4-bit binary form.
- E.g. 146 in BCD is

  0001 0100 0110



Decimal Digit	BCD 8 4 2 1
0	0 0 0 0
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
8	1 0 0 0
9	1 0 0 1

2.6.1 Binary Coded Decimal

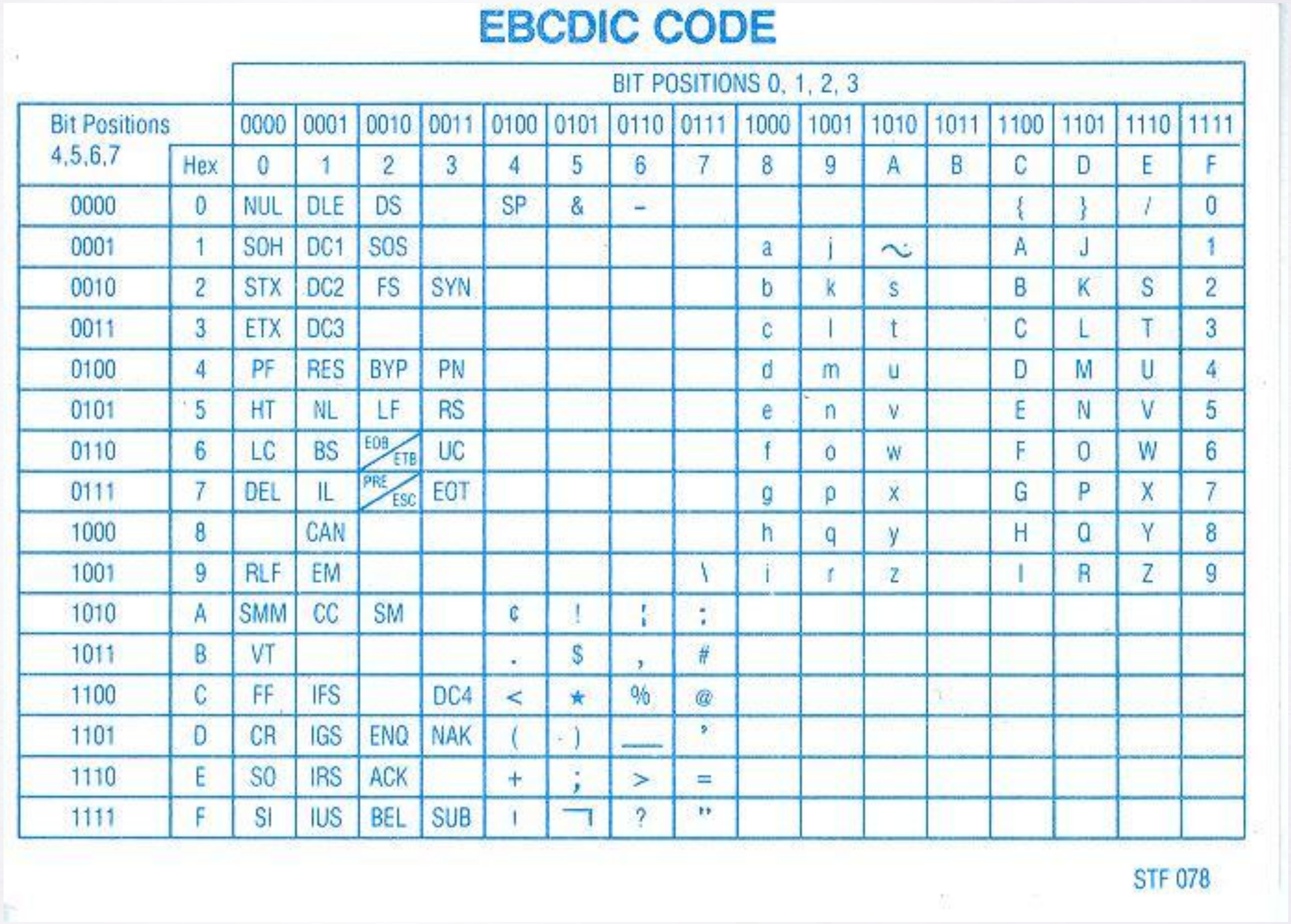
- Unpacked BCD: high order nibbles are padded with zeros.
  - Example: 146: 000000001 00000100 00000110
  - Unpacked BCD is wasteful.
- Packed BCD: stores two digits per byte.
  - Sign is stored at the end:
    1111: unsigned
    1100: positive
    1101: negative
  - Example: -146 would be stored as 00010100 01101101
  - Pad high-order nibble with zeros if necessary (e.g. -61)
- Disadvantage of BCD:
  - Many bits go to waste (1010 to 1111 not used)
- Advantage of BCD:
  - Can represent decimal numbers accurately.

2.6.1 Binary Coded Decimal

- Zoned decimal format BCD:
  - Same as for unpacked BCD, except that the high-order nibbles store a code (specifying the format at either EBCDIC: 1111 or ASCII: 0011)
  - High-order nibble of the last byte is the sign (1100: positive, 1101: negative)
  - Example: +146 in EBCDIC zoned decimal format is 11110001 11110100 11000110
  - Example: +146 in ASCII zoned decimal format is 00110001 00110100 11000110
- Example: Represent -1265 using packed BCD and EBCDIC zoned decimal.
  - Packed BCD: 00000001 00100110 01011101
  - EBCD: 11110001 11110010 11110110 11010101

2.6.2 EBCDIC

- Early version of BCD on IBM systems used 6-bit representation for characters and numbers (very limited, e.g. no lowercase letters).
- 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 still used by IBM mainframes today.
- Bytes in EBCDIC are in zone-digit form, e.g. character ‘a’ is 1000 0001 and digit ‘3’ is 1111 0011.



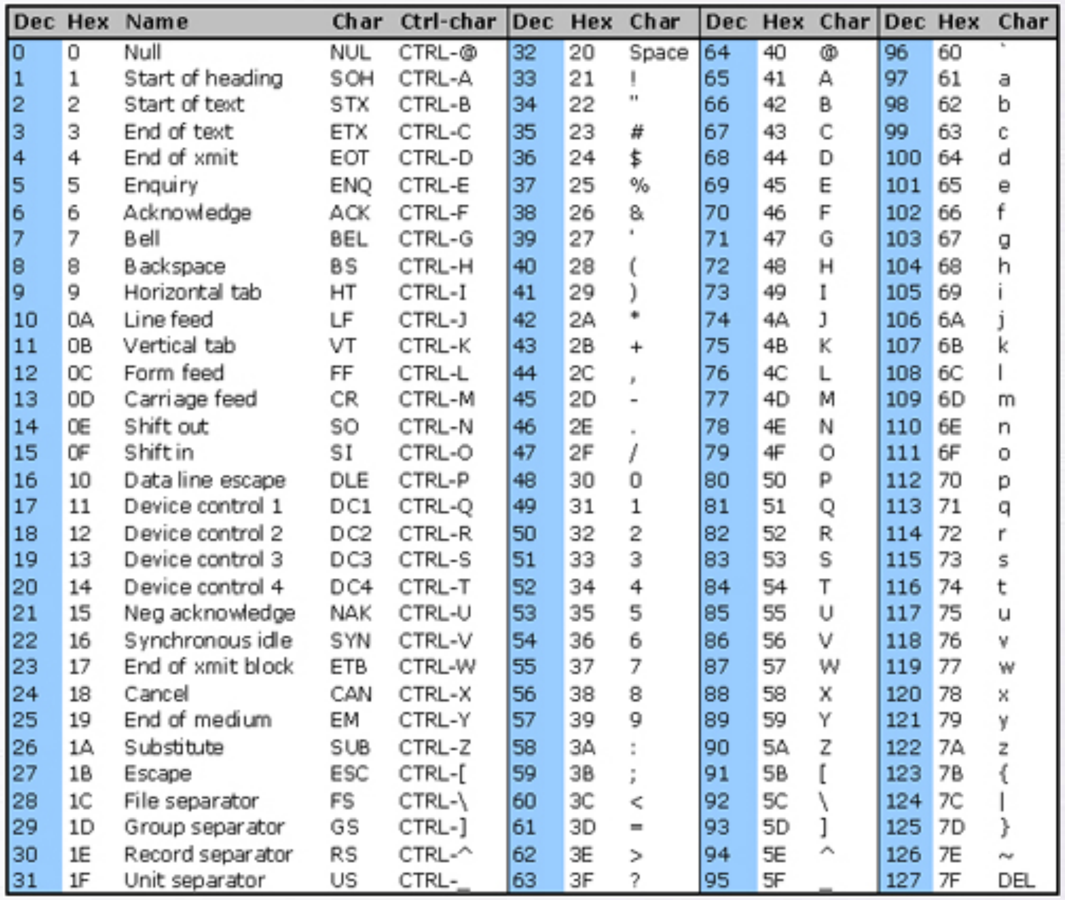
EBCDIC CODE

Bit Positions 4,5,6,7		BIT POSITIONS 0, 1, 2, 3															
		0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
	Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0000	0	NUL	DLE	DS		SP	&	-						{	}	/	0
0001	1	SOH	DC1	SOS						a	j	~		A	J		1
0010	2	STX	DC2	FS	SYN					b	k	s		B	K	S	2
0011	3	ETX	DC3							c	l	t		C	L	T	3
0100	4	PF	RES	BYP	PN					d	m	u		D	M	U	4
0101	5	HT	NL	LF	RS					e	n	v		E	N	V	5
0110	6	LC	BS	EOB / ETB	UC					f	o	w		F	O	W	6
0111	7	DEL	IL	PRE / ESC	EOT					g	p	x		G	P	X	7
1000	8		CAN							h	q	y		H	Q	Y	8
1001	9	RLF	EM						\	i	r	z		I	R	Z	9
1010	A	SMM	CC	SM		¢	!	¦	:								
1011	B	VT				.	$	,	#								
1100	C	FF	IFS		DC4	<	*	%	@								
1101	D	CR	IGS	ENQ	NAK	(	)	___	'								
1110	E	SO	IRS	ACK		+	;	>	=								
1111	F	SI	IUS	BEL	SUB	|	¬	?	"								

STF 078

2.6.3 ASCII

- Other computer manufacturers chose the 7-bit ASCII (American Standard Code for Information Interchange).
- Until recently, ASCII was the dominant character code outside the IBM mainframe world.
- The 8th bit was intended to be used for parity.



Dec	Hex	Name	Char	Ctrl-char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	Null	NUL	CTRL-@	32	20	Space	64	40	@	96	60	`
1	1	Start of heading	SOH	CTRL-A	33	21	!	65	41	A	97	61	a
2	2	Start of text	STX	CTRL-B	34	22	"	66	42	B	98	62	b
3	3	End of text	ETX	CTRL-C	35	23	#	67	43	C	99	63	c
4	4	End of xmit	EOT	CTRL-D	36	24	$	68	44	D	100	64	d
5	5	Enquiry	ENQ	CTRL-E	37	25	%	69	45	E	101	65	e
6	6	Acknowledge	ACK	CTRL-F	38	26	&	70	46	F	102	66	f
7	7	Bell	BEL	CTRL-G	39	27	'	71	47	G	103	67	g
8	8	Backspace	BS	CTRL-H	40	28	(	72	48	H	104	68	h
9	9	Horizontal tab	HT	CTRL-I	41	29	)	73	49	I	105	69	i
10	0A	Line feed	LF	CTRL-J	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	VT	CTRL-K	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	FF	CTRL-L	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage feed	CR	CTRL-M	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	SO	CTRL-N	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	SI	CTRL-O	47	2F	/	79	4F	O	111	6F	o
16	10	Data line escape	DLE	CTRL-P	48	30	0	80	50	P	112	70	p
17	11	Device control 1	DC1	CTRL-Q	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	DC2	CTRL-R	50	32	2	82	52	R	114	72	r
19	13	Device control 3	DC3	CTRL-S	51	33	3	83	53	S	115	73	s
20	14	Device control 4	DC4	CTRL-T	52	34	4	84	54	T	116	74	t
21	15	Neg acknowledge	NAK	CTRL-U	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	SYN	CTRL-V	54	36	6	86	56	V	118	76	v
23	17	End of xmit block	ETB	CTRL-W	55	37	7	87	57	W	119	77	w
24	18	Cancel	CAN	CTRL-X	56	38	8	88	58	X	120	78	x
25	19	End of medium	EM	CTRL-Y	57	39	9	89	59	Y	121	79	y
26	1A	Substitute	SUB	CTRL-Z	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	ESC	CTRL-[	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	FS	CTRL-\	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	GS	CTRL-]	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	RS	CTRL-^	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	US	CTRL-_	63	3F	?	95	5F	_	127	7F	DEL

2.6.3 ASCII

- Parity: the most basic of all error-detection schemes.
- A parity bit is 1 or 0 depending on whether the sum of the other bits is odd or even.
- E.g. ASCII ‘A’ is 65_{10} or 100 0001, so the bit string transmitted would be 0100 0001.
- Parity can be used to detect only single-bit errors.

2.6.4 Unicode

- Many of today's systems embrace Unicode, a 16-bit system that can encode the characters of every language in the world.
- Downward compatible with ASCII.
- The Unicode codespace is divided into six parts. The first part is for Western alphabet codes, including English, Greek, and Russian.

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

2.8 Error Detection and Correction

- It is physically impossible for any data recording or transmission medium to be 100% perfect 100% of the time over its entire expected useful life.
- As more bits are packed onto a square centimeter of disk storage, as communications transmission speeds increase, the likelihood of error.
- Thus, error detection and correction is critical to accurate data transmission, storage and retrieval.
- Check digits, appended to the end of a long number, can provide some protection against data input errors.
  - The last characters of ISBNs and UP student numbers are check digits.

2.7.1 Cyclic Redundancy Check

- A cyclic redundancy check (CRC) is a type of checksum that is used in error detection for large blocks of data.
- Checksums and CRCs are examples of *systematic error detection*.
- In *systematic error detection* a group of error control bits is appended to the end of the block of transmitted data.
  - This group of bits is called a *syndrome*.
  - Original information is unchanged by the addition of the error-checking bits.
- CRCs are polynomials over the modulo 2 arithmetic field.

The mathematical theory behind modulo 2 polynomials is beyond our scope. However, we can easily work with it without knowing its theoretical underpinnings.

2.7.1 Cyclic Redundancy Check

- Modulo 2 arithmetic works like clock arithmetic.
- In clock arithmetic, if we add 2 hours to 11:00, we get 1:00.
- In modulo 2 arithmetic if we add 1 to 1, we get 0. The addition rules couldn't be simpler:

0 + 0 = 0	0 + 1 = 1
1 + 0 = 1	1 + 1 = 0

- Example: Find the sum of 1011_2 and 110_2 modulo 2.

2.7.1 Cyclic Redundancy Check

- Example: Find the quotient and remainder when 1111101 is divided by 1101 in modulo 2 arithmetic.
  - As with traditional division, we note that the dividend is divisible once by the divisor.
  - We place the divisor under the dividend and perform modulo 2 subtraction.

```
           1
     _______
1101)1111101
     1101
     ----
     0010
```

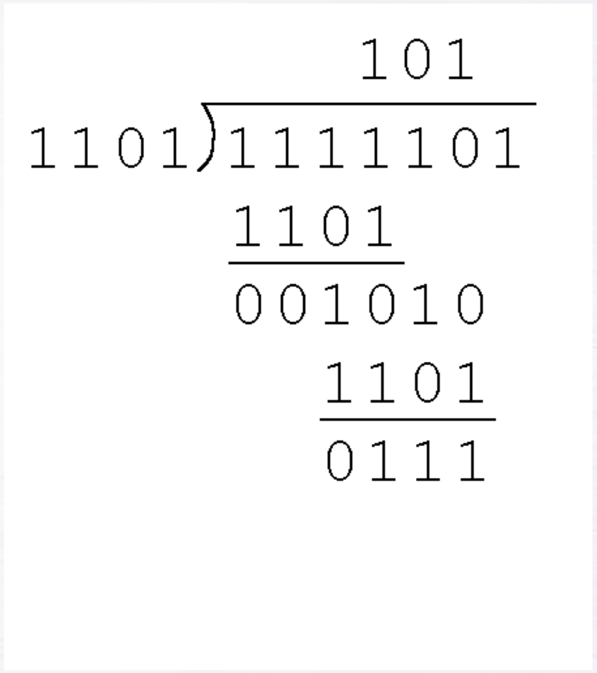
2.7.1 Cyclic Redundancy Check

- Find the quotient and remainder when 1111101 is divided by 1101 in modulo 2 arithmetic…
  - Now we bring down the next bit of the dividend.
  - We see that 00101 is not divisible by 1101. So we place a zero in the quotient.

```
        10
     _______
1101)1111101
     1101
     ____
     00101
```

2.7.1 Cyclic Redundancy Check

- Find the quotient and remainder when 1111101 is divided by 1101 in modulo 2 arithmetic…
  - 1010 is divisible by 1101 in modulo 2.
  - We perform the modulo 2 subtraction.



```
           101
     ________
1101)1111101
     1101
     ______
     001010
       1101
       ____
       0111
```

2.7.1 Cyclic Redundancy Check

- Find the quotient and remainder when 1111101 is divided by 1101 in modulo 2 arithmetic…
  - We find the quotient is 1011, and the remainder is 0010.
- This procedure is very useful to us in calculating CRC syndromes.

```
        1011
     _______
1101)1111101
     1101
     001010
       1101
       01111
        1101
        0010
```

Note: The divisor in this example corresponds to a modulo 2 polynomial: **X 3 + X 2 + 1.**

2.7.1 Cyclic Redundancy Check

- Suppose we want to transmit the information string: 1111101.
- The receiver and sender decide to use the (arbitrary) polynomial pattern, 1101.
- The information string is shifted left by one position less than the number of positions in the divisor.
- The remainder is found through modulo 2 division (at right) and added to the information string: 1111101000 + 111 = 1111101111.

```
        1011011
     __________
1101)1111101000
     1101
     ____
     001010
       1101
       ____
       01111
        1101
        ____
        001000
          1101
          ____
          01010
           1101
           ____
           0111
```

2.7.1 Cyclic Redundancy Check

- If no bits are lost or corrupted, dividing the received information string by the agreed upon pattern will give a remainder of zero.
- We see this is so in the calculation at the right.
- A remainder other than zero indicates an error has occurred.
- CRCs can be implemented effectively using lookup tables (instead of calculating the remainder with each byte)

```
             1011011
      ______________
1101 ) 1111101111
       1101
       ----
       001010
         1101
         ----
         01111
          1101
          ----
          001011
            1101
            ----
            01101
             1101
             ----
             0000
```

2.7 Error Detection and Correction

- Data transmission errors are easy to fix once an error is detected.
  - Just ask the sender to transmit the data again.
- In computer memory and data storage, however, this cannot be done.
  - Too often the only copy of something important is in memory or on disk.
- Thus, to provide data integrity over the long term, error *correcting* codes are required.
- Hamming codes and Reed-Solomon codes are two important error correcting codes.

2.7.2 Hamming Codes

- Hamming codes are code words formed by adding redundant check bits, or parity bits, to a data word.
- The memory word itself consists of *m* bits, but *r* redundant bits are added to allow for error detection and/or correction.
- The *Hamming distance* between two code words is the number of bits in which two code words differ (3 in example).

1	0	0	0	1	0	0	1
1	0	1	1	0	0	0	1

- The minimum Hamming distance for a code is the smallest Hamming distance between *all* pairs of words in the code.
- Example:
  - Assume memory with 2 data bits and 1 parity bit (appended at the end) that uses even parity (number of 1s in code word must be even).
  - Possible words in the code: 000, 011, 101, 110 (other possible bit patterns are invalid).
  - What is the minimum Hamming distance for this code?

2.7.2 Hamming Codes

- The minimum Hamming distance for a code, **D(min)**, determines its error detecting and error correcting capability.
- For a single-parity bit code:
  - Only single-bit errors can be detected, because D(min) is 2 (two bit errors can convert a valid code into another valid code).
- In general (for detection):
  - For any code word, *X,* to be interpreted as a different valid code word, *Y*, at least D(min) single-bit errors must occur in *X*.
  - Thus, to detect *k* (or fewer) single-bit errors, the code must have a Hamming distance of D(min) = *k* + 1.
  - Conversely, Hamming codes can always detect D(min) – 1 errors.
- For correction:
  - Hamming codes can correct $\left\lfloor \frac{D(\min) - 1}{2} \right\rfloor$ errors.
  - Or, the Hamming distance of a code must be at least 2*k* + 1 for it to be able to correct *k* errors.

Even with *k* errors, the word will still be closest to the original word.

Example 2.41

- Suppose we have the following code:

  0 0 0 0 0

  0 1 0 1 1

  1 0 1 1 0

  1 1 1 0 1

- What is D(min)?
  - Answer: D(min) = 3
- How many single-bit errors can be detected?
  - Answer: up to 2 single-bit errors
- How many single-bit errors can be corrected?
  - Answer: 1 single-bit errors can be corrected
- What is the correct code of the invalid code word 10000?
  - Difference vector to all code words: [1, 4, 2, 3], so correction is 00000 (closest code word). Assumption: the minimum number of possible errors has occurred. Correction might not be correct!
- What is the correct code of the invalid code word 11000?

Distance vector: [2, 3, 3, 2]. Cannot make the correction.

2.7.2 Hamming Codes

- Suppose we have a set of *n*-bit code words consisting of *m* data bits and *r* (redundant) check bits.
- How many legal code words?
  - Answer: 2^m
- How many illegal code words at a Hamming distance of 1 from each valid code word (i.e. bit strings with single-bit errors)?
  - Answer: *n* (an error can occur in any of the *n* bit positions)
- How many total bit patterns (valid and invalid) possible?
  - Answer: 2^n or $2^{(m+r)}$
- For each valid codeword, we have (*n*+1) bit patterns (1 legal and *n* illegal). This gives us the inequality:

$$(n + 1) \times 2^m \leq 2^n$$

2.7.2 Hamming Codes

- From previous slide:

$$(n + 1) \times 2^{m} \leq 2^{n}$$

- Because $n = m + r$, we can rewrite the inequality as:

$$(m + r + 1) \times 2^{m} \leq 2^{m+r}$$

or

$$(m + r + 1) \leq 2^{r}$$

- This inequality gives us a lower limit on the number of check bits that we need to construct a code with *m* data bits and *r* check bits that corrects all single-bit errors.
- For example, say we have data words of length 4. What is the minimum number of check bits needed for correcting single-bit errors?

  $(4 + r + 1) \leq 2^{r}$

  r must be greater than or equal to 3.
- We should always use the smallest value of *r* that makes the inequality true.

2.8 Error Detection and Correction

- For example, say we have data words of length 8. What is the minimum number of check bits needed for correcting single-bit errors?

  $(8 + r + 1) \leq 2^r$

  r must be greater than or equal to 4.
- How long will the resulting code words be?

  Answer: 12
- So how do we assign values to check bits so that we can achieve detection and correction?

The Hamming Algorithm

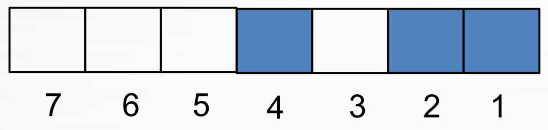
1. Determine the number of check bits, r, necessary for the code and then number the n bits, right to left, starting with 1 (not 0).
2. Each bit whose bit number is a power of 2 is a parity bit (positions 1,2,4,8,etc) and the others are data bits (positions 3,5,7,9,10,etc).
3. Each parity bit calculates the parity of some of the bits in the code word. The position of the parity bit determines the sequence of bits that it alternately checks and skips:

Position 1: check 1 bit, skip 1 bit, check 1 bit, skip 1 bit, etc. (1, 3, 5, 7, 9, …)
Position 2: check 2 bits, skip 2 bits, check 2 bits, etc. (2,3,6,7,10,11,…)
Position 4: check 4 bits, skip 4 bits, check 4 bits, etc. (4,5,6,7,12,13,14,15,20,...)
…and so on.

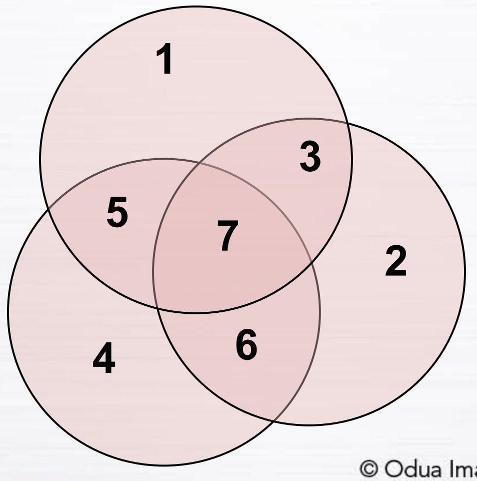
1. Set each parity bit to 1 if the number of 1's in the positions it checks is odd, or to 0 if the number of 1's in the position it checks is even (even parity).

2.7.2 Hamming Codes

- Consider the example with words of length 4, with 3 check bits.
- Code words will be of length 7 (white cells are data and green cells are parity bits):



  - The set of bits using parity bit 1: 1, 3, 5, 7 (check 1, skip 1, …)
  - The set of bits using parity bit 2: 2, 3, 6, 7 (check 2, skip 2, …)
  - The set of bits using parity bit 4: 4, 5, 6, 7 (check 4)



Notice that the parity bits (1, 2, 4) are each only in one set.

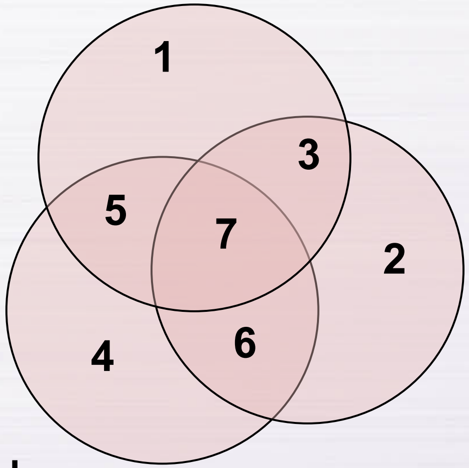
Bit position 7 is in all three sets.
Using this system, if a bit changes, it can be deduced which one changed.

2.7.2 Hamming Codes

- Example: Determine the Hamming code word for 4-bit data 1101.
- Code words will be of length 7 (white cells are data and green cells are parity bits):

1	1	0	🟦	1	🟦	🟦
7	6	5	4	3	2	1

  - The set of bits using parity bit 1: 1, 3, 5, 7 (check 1, skip 1, …)
  - The set of bits using parity bit 2: 2, 3, 6, 7 (check 2, skip 2, …)
  - The set of bits using parity bit 4: 4, 5, 6, 7 (check 4)
- Determine the parity bit values:
  - Parity bit 1: value is set to 0
  - Parity bit 2: value is set to 1
  - Parity bit 4: value is set to 0
- Hamming code word: 1100110
- How to determine where an error occurred?
  - Add the positions of parity bits that produce the error!



2.7.2 Hamming Codes

- What is the Hamming code of data word 11010110?
- Data word of length 8 requires 4 parity bits.
- Which will be the check (parity) bits?

				P				P		P	P
12	11	10	9	8	7	6	5	4	3	2	1

- What is the full Hamming code?

1	1	0	1	1	0	1	1	1	0	0	1
12	11	10	9	8	7	6	5	4	3	2	1

2.7.2 Hamming Codes

1	1	0	1	1	0	1	0	1	0	0	1
12	11	10	9	8	7	6	5	4	3	2	1

- There is an error in the code word above. Correct it.
  - Bit 1 checks 1, 3, 5, 7, 9, and 11. *This is incorrect as we have a total of 3 ones (which is not even parity).*
  - Bit 2 checks bits 2, 3, 6, 7, 10, and 11. The parity is correct.
  - Bit 4 checks bits 4, 5, 6, 7, and 12. *This parity is incorrect, as we 3 ones.*
  - Bit 8 checks bit 8, 9, 10, 11, and 12. This parity is correct.
  - The common elements in the two sets are 5 and 7, but 7 is part of bit 2 check, so cannot be the problem.
  - Therefore, the bit that changed was 5, so flip the bit in position 5.
  - Alternatively, simply add 1 + 4 = 5, and flip bit 5.

Example 2.43

- Use the Hamming algorithm to find all code words for a 3-bit memory word, assuming odd parity.

2.7.3 Reed Solomon

- Burst errors: multiple adjacent bits are damaged, e.g a scratch on a CD.
- Since Hamming code operates at the bit level, it is useless for burst errors.
- Need an error-correcting code that operates on a block level.
- One such technique is to use Reed-Solomon codes (which operator on the character level, not bit level).

Chapter 2 Conclusion

- Computers store data in the form of bits, bytes, and words using the binary numbering system.
- Hexadecimal numbers are formed using four-bit groups called nibbles.
- Signed integers can be stored in one’s complement, two’s complement, or signed magnitude representation.
- Floating-point numbers are usually coded using the IEEE 754 floating-point standard.

Chapter 2 Conclusion

- Floating-point operations are not necessarily commutative or distributive.
- Character data is stored using ASCII, EBCDIC, or Unicode.
- Error detecting and correcting codes are necessary because we can expect no transmission or storage medium to be perfect.
- CRC, Reed-Solomon, and Hamming codes are three important error control codes.