Review Slides

Data Representation

2.1 Introduction

- A *bit* is the most basic unit of information in a computer.
  - It is a state of "on" or "off" in a digital circuit.
  - Sometimes these states are "high" or "low" voltage instead of "on" or "off.."
- A *byte* is a group of eight 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.

2.1 Introduction

- A *word* is a contiguous group of bytes.
  - Words can be any number of bits or bytes.
  - Word sizes of 16, 32, or 64 bits are most common.
  - In a word-addressable system, a word is the smallest addressable unit of storage.
- A group of four bits is called a *nibble*.
  - Bytes, therefore, consist of two nibbles: a "high-order nibble," and a "low-order" nibble.

2.2 Positional Numbering Systems

- Bytes store numbers using the position of each bit to represent a power of 2.
  - The binary system is also called the base-2 system.
  - Our decimal system is the base-10 system. It uses powers of 10 for each position in a number.
  - Any integer quantity can be represented exactly using any base (or *radix*).

2.2 Positional Numbering Systems

- The decimal number 947 in powers of 10 is:

$$9 \times 10^2 + 4 \times 10^1 + 7 \times 10^0$$

- The decimal number 5836.47 in powers of 10 is:

$$5 \times 10^3 + 8 \times 10^2 + 3 \times 10^1 + 6 \times 10^0 + 4 \times 10^{-1} + 7 \times 10^{-2}$$

2.2 Positional Numbering Systems

- The binary number 11001 in powers of 2 is:

$$1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$
$$= 16 + 8 + 0 + 0 + 1 = 25$$

- When the radix of a number is something other than 10, the base is denoted by a subscript.
  - Sometimes, the subscript 10 is added for emphasis:

$$11001_2 = 25_{10}$$

2.3 Converting Between Bases

- Because binary numbers are the basis for all data representation in digital computer systems, it is important that you become proficient with this radix system.
- Your knowledge of the binary numbering system will enable you to understand the operation of all computer components as well as the design of instruction set architectures.

2.3 Converting Between Bases

- In an earlier slide, we said that every integer value can be represented exactly using any radix system.
- There are two methods for radix conversion: the subtraction method and the division remainder method.
- The subtraction method is more intuitive, but cumbersome. It does, however reinforce the ideas behind radix mathematics.

2.3 Converting Between Bases

- Suppose we want to convert the decimal number 190 to base 3.
  - We know that $3^5 = 243$ so our result will be less than six digits wide. The largest power of 3 that we need is therefore $3^4 = 81$, and $81 \times 2 = 162$.
  - Write down the 2 and subtract 162 from 190, giving 28.

$$\begin{array}{r} 190 \\ -\ 162 \\ \hline 28 \end{array} = 3^4 \times 2$$

2.3 Converting Between Bases

- Converting 190 to base 3...
  - The next power of 3 is $3^3 = 27$. We'll need one of these, so we subtract 27 and write down the numeral 1 in our result.
  - The next power of 3, $3^2 = 9$, is too large, but we have to assign a placeholder of zero and carry down the 1.

$$
\begin{array}{rl}
190 & \\
-\ 162 & = 3^4 \times 2 \\
\hline
28 & \\
-\ 27 & = 3^3 \times 1 \\
\hline
1 & \\
-\ 0 & = 3^2 \times 0 \\
\hline
1 &
\end{array}
$$

2.3 Converting Between Bases

- Converting 190 to base 3...
  - $3^1 = 3$ is again too large, so we assign a zero placeholder.
  - The last power of 3, $3^0 = 1$, is our last choice, and it gives us a difference of zero.
  - Our result, reading from top to bottom is:

    $190_{10} = 21001_3$

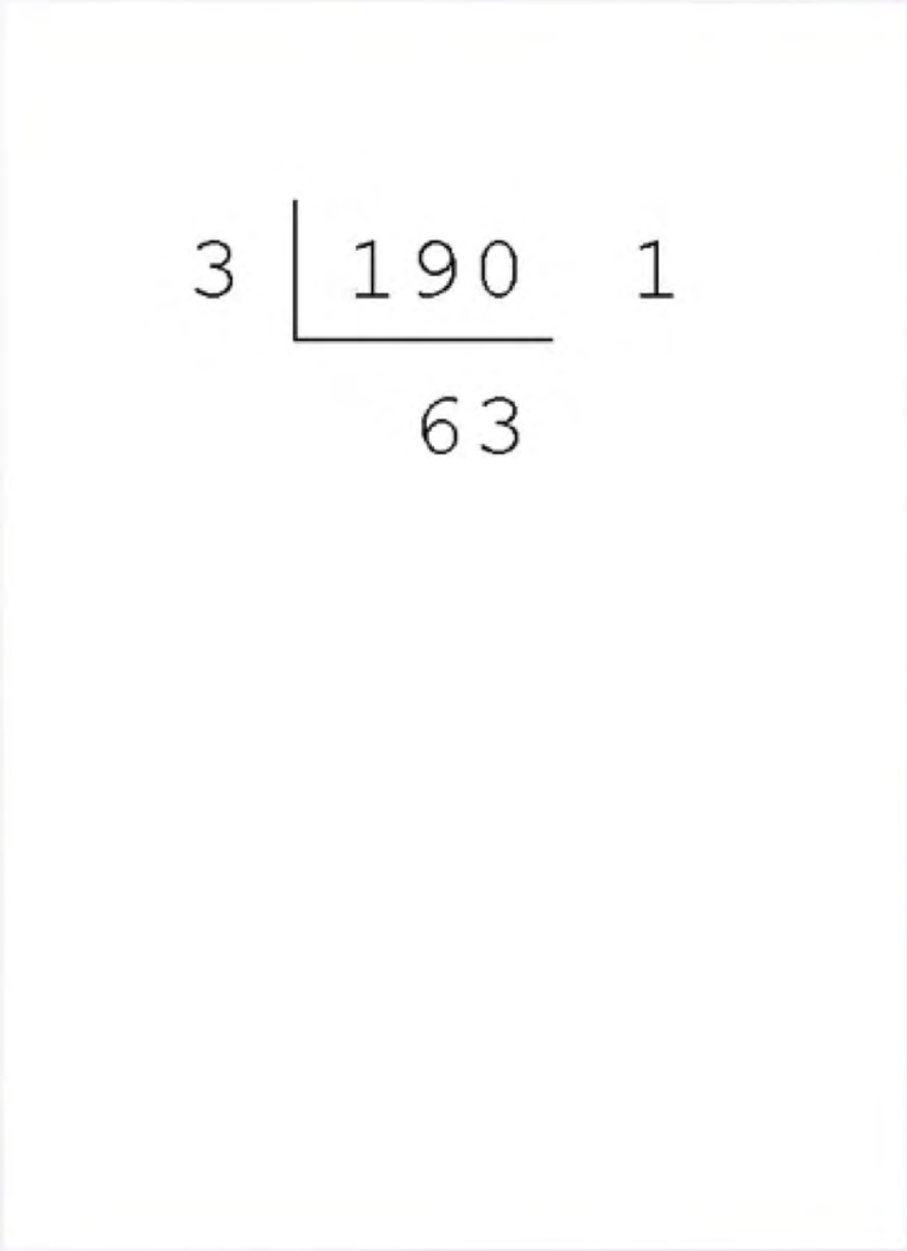
```
  190
- 162  = 3^4 × 2
-----
   28
-  27  = 3^3 × 1
-----
    1
-   0  = 3^2 × 0
-----
    1
-   0  = 3^1 × 0
-----
    1
-   1  = 3^0 × 1
-----
    0
```

2.3 Converting Between Bases

- Another method of converting integers from decimal to some other radix uses division.
- This method is mechanical and easy.
- It employs the idea that successive division by a base is equivalent to successive subtraction by powers of the base.
- Let's use the division remainder method to again convert 190 in decimal to base 3.

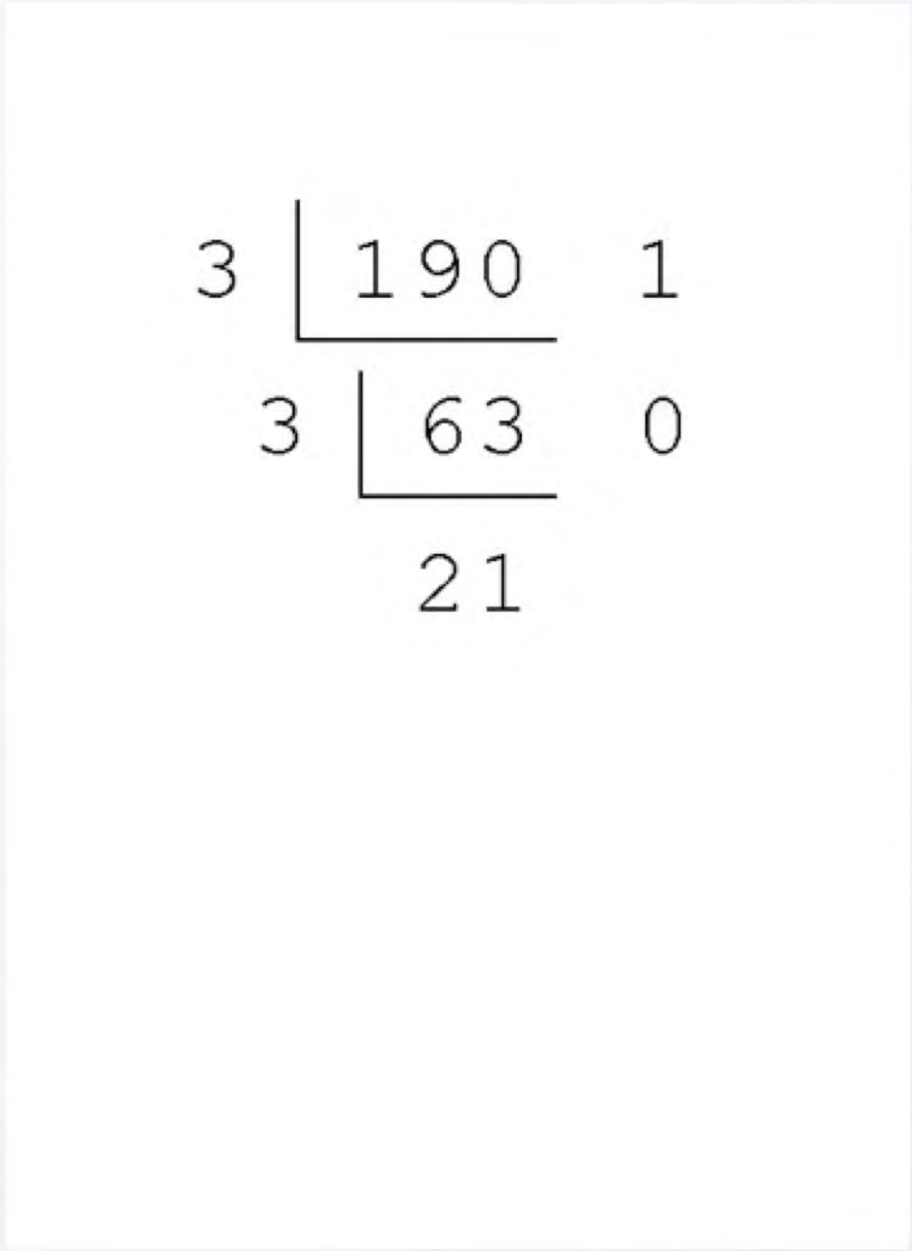
2.3 Converting Between Bases

- Converting 190 to base 3...
  - First we take the number that we wish to convert and divide it by the radix in which we want to express our result.
  - In this case, 3 divides 190 63 times, with a remainder of 1.
  - Record the quotient and the remainder.



2.3 Converting Between Bases

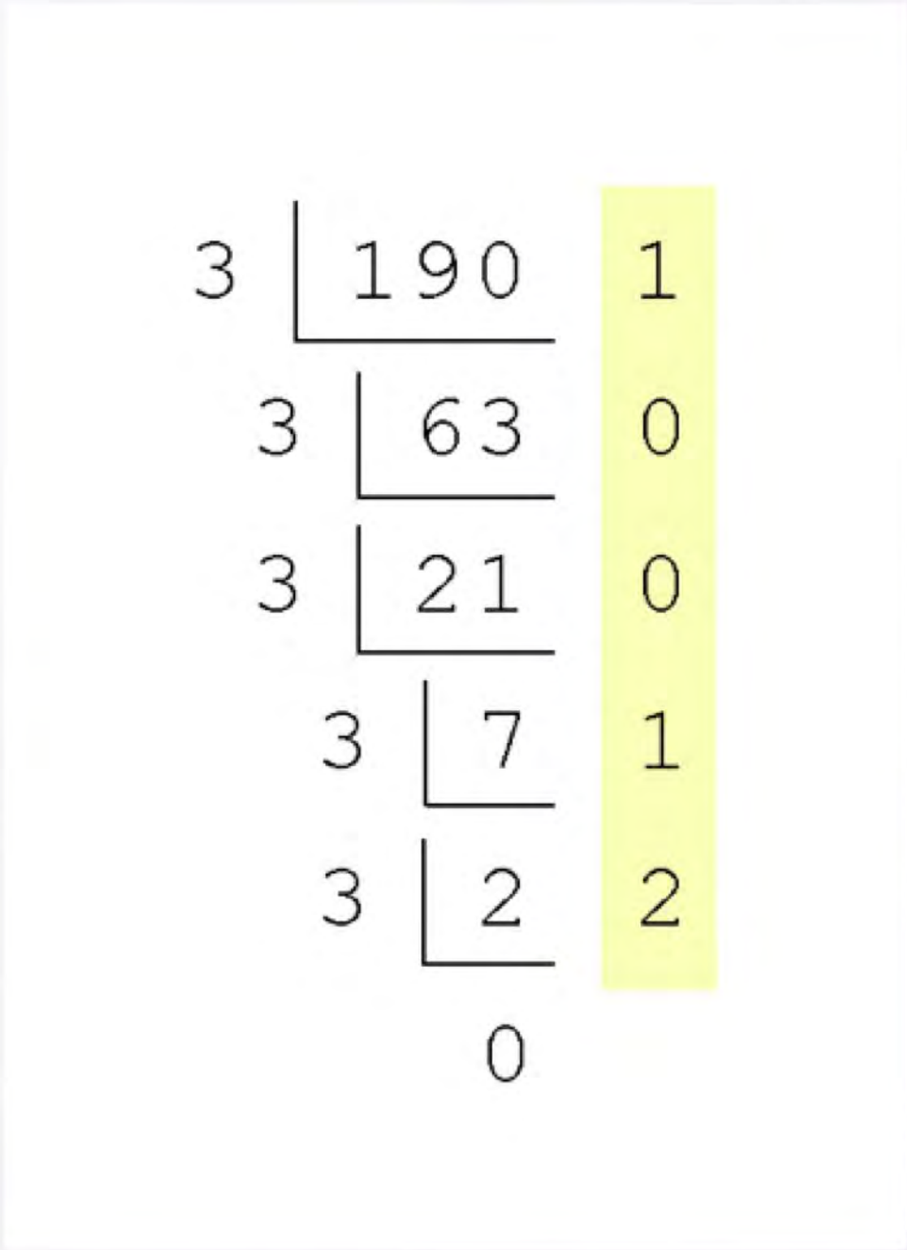
- Converting 190 to base 3...
  - 63 is evenly divisible by 3.
  - Our remainder is zero, and the quotient is 21.



2.3 Converting Between Bases

- Converting 190 to base 3...
  - Continue in this way until the quotient is zero.
  - In the final calculation, we note that 3 divides 2 zero times with a remainder of 2.
  - Our result, reading from bottom to top is:

$$190_{10} = 21001_3$$



2.3 Converting Between Bases

- The binary numbering system is the most important radix system for digital computers.
- However, it is difficult to read long strings of binary numbers -- and even a modestly-sized decimal number becomes a very long binary number.
  - For example: $11010100011011_2 = 13595_{10}$
- For compactness and ease of reading, binary values are usually expressed using the hexadecimal, or base-16, numbering system.

2.3 Converting Between Bases

- The hexadecimal numbering system uses the numerals 0 through 9 and the letters A through F.
  - The decimal number 12 is C_{16}.
  - The decimal number 26 is $1A_{16}$.
- It is easy to convert between base 16 and base 2, because $16 = 2^4$.
- Thus, to convert from binary to hexadecimal, all we need to do is group the binary digits into groups of four.

A group of four binary digits is called a hextet

2.3 Converting Between Bases

- Using groups of hextets, the binary number 11010100011011_2 ($= 13595_{10}$) in hexadecimal is:

0011	0101	0001	1011
3	5	1	B

If the number of bits is not a multiple of 4, pad on the left with zeros.

- Octal (base 8) values are derived from binary by using groups of three bits ($8 = 2^3$):

011	010	100	011	011
3	2	4	3	3

Octal was very useful when computers used six-bit words.

Bases

Decimal	Binary	Octal	Hexadecimal
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

2.3 Converting Between Bases

- Fractional values can be approximated in all base systems.
- Unlike integer values, fractions do not necessarily have exact representations under all radices.
- The quantity ½ is exactly representable in the binary and decimal systems, but is not in the ternary (base 3) numbering system.

2.3 Converting Between Bases

- Fractional decimal values have nonzero digits to the right of the decimal point.
- Fractional values of other radix systems have nonzero digits to the right of the *radix point*.
- Numerals to the right of a radix point represent negative powers of the radix:

$$0.47_{10} = 4 \times 10^{-1} + 7 \times 10^{-2}$$
$$0.11_2 = 1 \times 2^{-1} + 1 \times 2^{-2}$$
$$= \frac{1}{2} + \frac{1}{4}$$
$$= 0.5 + 0.25 = 0.75$$

2.3 Converting Between Bases

- As with whole-number conversions, you can use either of two methods: a subtraction method or an easy multiplication method.
- The subtraction method for fractions is identical to the subtraction method for whole numbers. Instead of subtracting positive powers of the target radix, we subtract negative powers of the radix.
- We always start with the largest value first, n^{-1}, where n is our radix, and work our way along using larger negative exponents.

2.3 Converting Between Bases

- The calculation to the right is an example of using the subtraction method to convert the decimal 0.8125 to binary.
  - Our result, reading from <u>top to bottom</u> is:

$$0.8125_{10} = 0.1101_2$$

  - Of course, this method works with any base, not just binary.

$$\begin{array}{rl} 0.8125 & \\ -\ 0.5000 & = 2^{-1} \times 1 \\ \hline 0.3125 & \\ -\ 0.2500 & = 2^{-2} \times 1 \\ \hline 0.0625 & \\ -\ 0 & = 2^{-3} \times 0 \\ \hline 0.0625 & \\ -\ 0.0625 & = 2^{-4} \times 1 \\ \hline 0 & \end{array}$$

2.3 Converting Between Bases

- Using the multiplication method to convert the decimal 0.8125 to binary, we multiply by the radix 2.
  - The first product carries into the units place.

```
  .8125
×    2
------
1.6250
```

2.3 Converting Between Bases

- Converting 0.8125 to binary . . .
  - Ignoring the value in the units place at each step, continue multiplying each fractional part by the radix.

```
  .8125
 ×   2
1.6250

  .6250
 ×   2
1.2500

  .2500
 ×   2
0.5000
```

2.3 Converting Between Bases

- Converting 0.8125 to binary . . .
  - You are finished when the product is zero, or until you have reached the desired number of binary places.
  - Our result, reading from top to bottom is:

    $0.8125_{10} = 0.1101_2$
  - This method also works with any base. Just use the target radix as the multiplier.

```
  .8125
 ×   2
1.6250

  .6250
 ×   2
1.2500

  .2500
 ×   2
0.5000

  .5000
 ×   2
1.0000
```

Convert Base 6 to Base 10

$$123.45_6 = ???.??_{10}$$

$$123 = 1*6^2{}_{10}\,[1*36_{10}] +$$
$$2*6^1{}_{10}\,[2*6_{10}] +$$
$$3*6^0{}_{10}\,[3*1_{10}] =$$
$$51_{10}$$

$$0.45 = 4*6^{-1}{}_{10}\,[4*1/6_{10}] +$$
$$5*6^{-2}{}_{10}\,[5*1/36_{10}] =$$
$$.80\overline{5}_{10}$$

$$123.45_6 = 51.80\overline{5}_{10}$$

Convert Base 10 to Base 6

$754.94_{10} = ???.??_{6}$

754 / 6 = 125 remainder 4

125 / 6 = 20 remainder 5

20 / 6 = 3 remainder 2

3 / 6 = 0 remainder 3

$754_{10} = 3*6^3 + 2*6^2 + 5*6^1 + 4*6^0$

3254_{6}

Convert Base 10 to Base 6

$.94_{10} = .???_6$

.94 * 6 = 5.64 —> Keep 5

.64 * 6 = 3.84 —> Keep 3

.84 * 6 = 5.04 —> Keep 5

.04 * 6 = 0.24 —> Keep 0

.24 * 6 = 1.44 —> Keep 1

.44 * 6 = 2.64 —> Keep 2

.64 * 6 = 3.84 —> We're repeating now

$$.94_6 = 5*6^{-1} + 3*6^{-2} + 5*6^{-3} + 0*6^{-4} + 1*6^{-5} + 2*6^{-6}$$

$$3254.5\overline{35012}_6$$

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 0x12345678.
- 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

2.4 Signed Integer Representation

- The conversions we have so far presented have involved only unsigned numbers.
- To represent signed integers, computer systems allocate the high-order 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 (but this can be interpreted different ways)

2.4 Signed Integer Representation

- There are three ways in which signed binary integers may be expressed:
  - Signed magnitude
  - One's complement
  - Two's complement
- 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.

2.4 Signed Integer Representation

- For example, in 8-bit signed magnitude representation:
  - +3 is: `00000011`
  - - 3 is: `10000011`
- Computers perform arithmetic operations on signed magnitude numbers in much the same way as humans carry out pencil and paper arithmetic.
  - Humans often ignore the signs of the operands while performing a calculation, applying the appropriate sign after the calculation is complete.

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.

```
0   1001011
0 + 0101110
```

2.4 Signed Integer Representation

- Example:
  - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- Just as in decimal arithmetic, we find the sum starting with the rightmost bit and work left.

```
  1001011
+ 0101110
-------
  1
```

2.4 Signed Integer Representation

- Example:
  - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- In the second bit, we have a carry, so we note it above the third bit.

```
          1
0   1001011
0 + 0101110
-----------
       01
```

2.4 Signed Integer Representation

- Example:
  - Using signed magnitude binary arithmetic, find the sum of 75 and 46.
- The third and fourth bits also give us carries.

```
        1 1 1
0   1001011
0 + 0101110
-----------
      1001
```

2.4 Signed Integer Representation

- Example:
  - Using signed magnitude binary arithmetic, find the sum of 75 and 46.

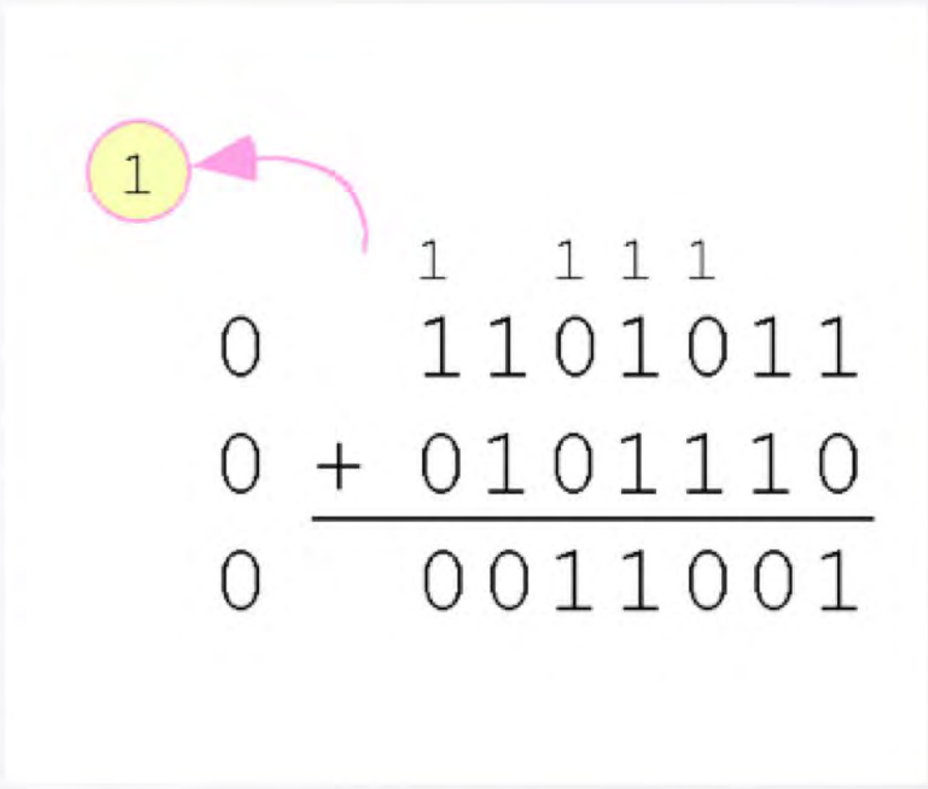
```
      1 1 1
0   1001011
0 + 0101110
-----------
0   1111001
```

- Once we have worked our way through all eight bits, we are done.

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.



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.

```
    1 1
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.

```
          0 2     0 2
0    0 1 0 1 1 1 0
1 + 0 0 1 1 0 0 1
-----------------
0    0 0 1 0 1 0 1
```

- 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.

2.4 Signed Integer Representation

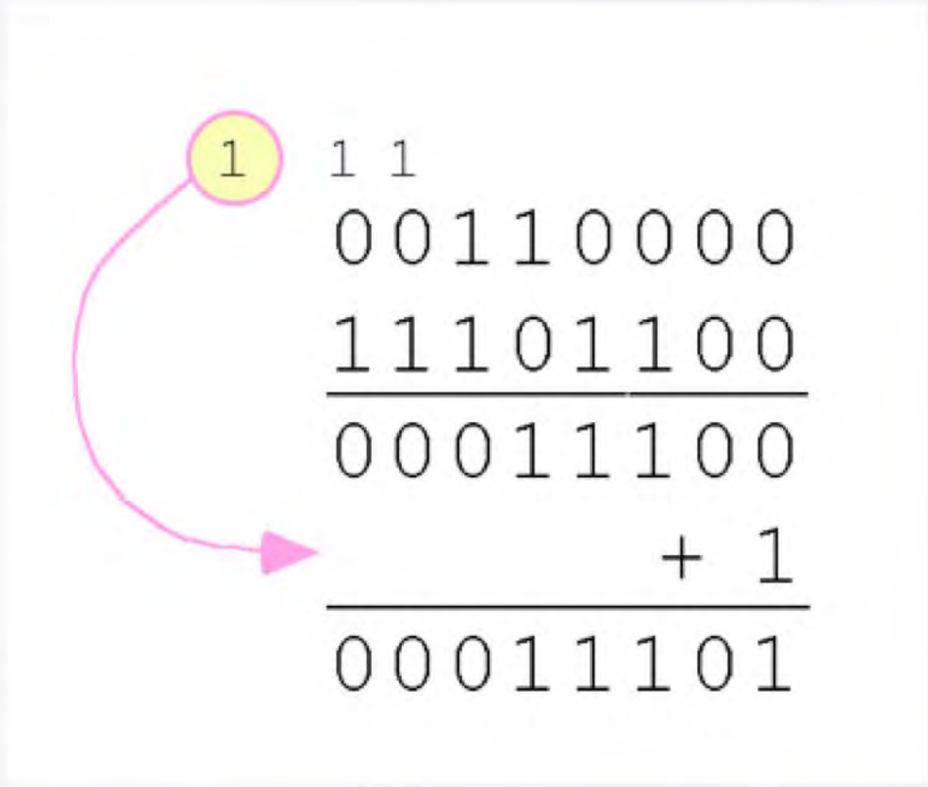
- In complement systems, negative values are represented by some difference between a number and its base.
- The *diminished radix complement* of a non-zero number *N* in base *r* with *d* digits is $(r^d - 1) - N$
- In the binary system, this gives us *one's complement*. It amounts to little more than flipping the bits of a binary number.

2.4 Signed Integer Representation

- For example, using 8-bit one's complement representation:
  + 3 is: `00000011`
  - 3 is: `11111100`
- In one's complement representation, as with signed magnitude, negative values are indicated by a 1 in the high order bit.
- Complement systems are useful because they eliminate the need for subtraction. The difference of two values is found by adding the minuend to the complement of the subtrahend.

2.4 Signed Integer Representation

- With one's complement addition, the carry bit is "carried around" and added to the sum.
  - Example: Using one's complement binary arithmetic, find the sum of 48 and - 19



We note that 19 in binary is `00010011`,
so -19 in one's complement is: `11101100`.

2.4 Signed Integer Representation

- Although the “end carry around” adds some complexity, one’s complement is simpler to implement than signed magnitude.
- But it still has the disadvantage of having two different representations for zero: positive zero and negative zero.
- Two’s complement solves this problem.
- Two’s complement is the radix complement of the binary numbering system; the *radix complement* of a non-zero number N in base r with d digits is $r^d - N$.

2.4 Signed Integer Representation

- To express a value in two's complement representation:
  - If the number is positive, just convert it to binary and you're done.
  - If the number is negative, find the one's complement of the number and then add 1.
- Example:
  - In 8-bit binary, 3 is: `00000011`
  - -3 using one's complement representation is: `11111100`
  - Adding 1 gives us -3 in two's complement form: `11111101`.

2.4 Signed Integer Representation

- With two's complement arithmetic, all we do is add our two binary numbers. Just discard any carries emitting from the high order bit.

– Example: Using one's complement binary arithmetic, find the sum of 48 and - 19.

```
  1          1 1
          00110000
        + 11101101
        ----------
          00011101
```

We note that 19 in binary is: **00010011**,
so -19 using one's complement is: **11101100**,
and -19 using two's complement is: **11101101**.

2.4 Signed Integer Representation

- Lets compare our representations:

Decimal	Binary (for absolute value)	Signed Magnitude	One's Complement
2	00000010	00000010	00000010
−2	00000010	10000010	11111101
100	01100100	01100100	01100100
−100	01100100	11100100	10011011

Decimal	Binary (for absolute value)	Two's Complement	Excess-127
2	00000010	00000010	10000001
−2	00000010	11111110	01111101
100	01100100	01100100	11100011
−100	01100100	10011100	00011011

2.4 Signed Integer Representation

- When we use any finite number of bits to represent a number, we always run the risk of the result of our calculations becoming too large or too small to be stored in the computer.
- While we can't always prevent overflow, we can always *detect* overflow.
- In complement arithmetic, an overflow condition is easy to detect.

2.4 Signed Integer Representation

- Example:
  - Using two's complement binary arithmetic, find the sum of 107 and 46.

```
  1 1   1 1 1
  01101011
+ 00101110
----------
  10011001
```

- We see that the nonzero carry from the seventh bit *overflows* into the sign bit, giving us the erroneous result: 107 + 46 = -103.

But overflow into the sign bit does not always mean that we have an error.

2.4 Signed Integer Representation

- Example:
  - Using two's complement binary arithmetic, find the sum of 23 and -9.
  - We see that there is carry into the sign bit and carry out. The final result is correct: 23 + (-9) = 14.

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

Rule for detecting signed two's complement overflow: When the "carry in" and the "carry out" of the sign bit differ, overflow has occurred. If the carry into the sign bit equals the carry out of the sign bit, no overflow has occurred.

2.4 Signed Integer Representation

- Signed and unsigned numbers are both useful.
  - For example, memory addresses are always unsigned.
- Using the same number of bits, unsigned integers can express twice as many "positive" values as signed numbers.
- Trouble arises if an unsigned value "wraps around."
  - In four bits: $1111 + 1 = 0000$.
- Good programmers stay alert for this kind of problem.

2.4 Signed Integer Representation

- Overflow and carry are tricky ideas.
- Signed number overflow means nothing in the context of unsigned numbers, which set a carry flag instead of an overflow flag.
- If a carry out of the leftmost bit occurs with an unsigned number, overflow has occurred.
- Carry and overflow occur independently of each other.

The table on the next slide summarizes these ideas.

2.4 Signed Integer Representation

Expression	Result	Carry?	Overflow?	Correct Result?
0100 + 0010	0110	No	No	Yes
0100 + 0110	1010	No	Yes	No
1100 + 1110	1010	Yes	No	Yes
1100 + 1010	0110	Yes	Yes	No

2.4 Signed Integer Representation

- We can do binary multiplication and division by 2 very easily using an *arithmetic shift* operation
- A *left arithmetic shift* inserts a 0 in for the rightmost bit and shifts everything else left one bit; in effect, it multiplies by 2
- A *right arithmetic shift* shifts everything one bit to the right, but copies the sign bit; it divides by 2
- Let's look at some examples.

2.4 Signed Integer Representation

Example:

Multiply the value 11 (expressed using 8-bit signed two's complement representation) by 2.

We start with the binary value for 11:

00001011 (+11)

We shift left one place, resulting in:

00010110 (+22)

The sign bit has not changed, so the value is valid.

To multiply 11 by 4, we simply perform a left shift twice.

2.4 Signed Integer Representation

Example:

Divide the value 12 (expressed using 8-bit signed two's complement representation) by 2.

We start with the binary value for 12:

00001100 (+12)

We shift left one place, resulting in:

00000110 (+6)

(Remember, we carry the sign bit to the left as we shift.)

To divide 12 by 4, we right shift twice.

Character Codes

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.

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.

The ASCII code

American Standard Code for Information Interchange

ASCII control characters

DEC	HEX	Simbolo ASCII	
00	00h	NULL	(carácter nulo)
01	01h	SOH	(inicio encabezado)
02	02h	STX	(inicio texto)
03	03h	ETX	(fin de texto)
04	04h	EOT	(fin transmisión)
05	05h	ENQ	(enquiry)
06	06h	ACK	(acknowledgement)
07	07h	BEL	(timbre)
08	08h	BS	(retroceso)
09	09h	HT	(tab horizontal)
10	0Ah	LF	(salto de linea)
11	0Bh	VT	(tab vertical)
12	0Ch	FF	(form feed)
13	0Dh	CR	(retorno de carro)
14	0Eh	SO	(shift Out)
15	0Fh	SI	(shift In)
16	10h	DLE	(data link escape)
17	11h	DC1	(device control 1)
18	12h	DC2	(device control 2)
19	13h	DC3	(device control 3)
20	14h	DC4	(device control 4)
21	15h	NAK	(negative acknowle.)
22	16h	SYN	(synchronous idle)
23	17h	ETB	(end of trans. block)
24	18h	CAN	(cancel)
25	19h	EM	(end of medium)
26	1Ah	SUB	(substitute)
27	1Bh	ESC	(escape)
28	1Ch	FS	(file separator)
29	1Dh	GS	(group separator)
30	1Eh	RS	(record separator)
31	1Fh	US	(unit separator)
127	20h	DEL	(delete)

ASCII printable characters

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
32	20h	espacio	64	40h	@	96	60h	`
33	21h	!	65	41h	A	97	61h	a
34	22h	"	66	42h	B	98	62h	b
35	23h	#	67	43h	C	99	63h	c
36	24h	$	68	44h	D	100	64h	d
37	25h	%	69	45h	E	101	65h	e
38	26h	&	70	46h	F	102	66h	f
39	27h	'	71	47h	G	103	67h	g
40	28h	(	72	48h	H	104	68h	h
41	29h	)	73	49h	I	105	69h	i
42	2Ah	*	74	4Ah	J	106	6Ah	j
43	2Bh	+	75	4Bh	K	107	6Bh	k
44	2Ch	,	76	4Ch	L	108	6Ch	l
45	2Dh	-	77	4Dh	M	109	6Dh	m
46	2Eh	.	78	4Eh	N	110	6Eh	n
47	2Fh	/	79	4Fh	O	111	6Fh	o
48	30h	0	80	50h	P	112	70h	p
49	31h	1	81	51h	Q	113	71h	q
50	32h	2	82	52h	R	114	72h	r
51	33h	3	83	53h	S	115	73h	s
52	34h	4	84	54h	T	116	74h	t
53	35h	5	85	55h	U	117	75h	u
54	36h	6	86	56h	V	118	76h	v
55	37h	7	87	57h	W	119	77h	w
56	38h	8	88	58h	X	120	78h	x
57	39h	9	89	59h	Y	121	79h	y
58	3Ah	:	90	5Ah	Z	122	7Ah	z
59	3Bh	;	91	5Bh	[	123	7Bh	{
60	3Ch	<	92	5Ch	\	124	7Ch	|
61	3Dh	=	93	5Dh	]	125	7Dh	}
62	3Eh	>	94	5Eh	^	126	7Eh	~
63	3Fh	?	95	5Fh	_			

theASCIIcode.com.ar

Extended ASCII characters

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
128	80h	Ç	160	A0h	á	192	C0h	└	224	E0h	Ó
129	81h	ü	161	A1h	í	193	C1h	┴	225	E1h	ß
130	82h	é	162	A2h	ó	194	C2h	┬	226	E2h	Ô
131	83h	â	163	A3h	ú	195	C3h	├	227	E3h	Ò
132	84h	ä	164	A4h	ñ	196	C4h	─	228	E4h	õ
133	85h	à	165	A5h	Ñ	197	C5h	┼	229	E5h	Õ
134	86h	å	166	A6h	ª	198	C6h	ã	230	E6h	µ
135	87h	ç	167	A7h	º	199	C7h	Ã	231	E7h	þ
136	88h	ê	168	A8h	¿	200	C8h	╚	232	E8h	Þ
137	89h	ë	169	A9h	®	201	C9h	╔	233	E9h	Ú
138	8Ah	è	170	AAh	¬	202	CAh	╩	234	EAh	Û
139	8Bh	ï	171	ABh	½	203	CBh	╦	235	EBh	Ù
140	8Ch	î	172	ACh	¼	204	CCh	╠	236	ECh	ý
141	8Dh	ì	173	ADh	¡	205	CDh	═	237	EDh	Ý
142	8Eh	Ä	174	AEh	«	206	CEh	╬	238	EEh	¯
143	8Fh	Å	175	AFh	»	207	CFh	¤	239	EFh	´
144	90h	É	176	B0h	░	208	D0h	ð	240	F0h	
145	91h	æ	177	B1h	▒	209	D1h	Ð	241	F1h	±
146	92h	Æ	178	B2h	▓	210	D2h	Ê	242	F2h	‗
147	93h	ô	179	B3h	│	211	D3h	Ë	243	F3h	¾
148	94h	ò	180	B4h	┤	212	D4h	È	244	F4h	¶
149	95h	ò	181	B5h	Á	213	D5h	ı	245	F5h	§
150	96h	û	182	B6h	Â	214	D6h	Í	246	F6h	÷
151	97h	ù	183	B7h	À	215	D7h	Î	247	F7h	¸
152	98h	ÿ	184	B8h	©	216	D8h	Ï	248	F8h	°
153	99h	Ö	185	B9h	╣	217	D9h	┘	249	F9h	¨
154	9Ah	Ü	186	BAh	║	218	DAh	┌	250	FAh	·
155	9Bh	ø	187	BBh	╗	219	DBh	█	251	FBh	¹
156	9Ch	£	188	BCh	╝	220	DCh	▄	252	FCh	³
157	9Dh	Ø	189	BDh	¢	221	DDh	¦	253	FDh	²
158	9Eh	×	190	BEh	¥	222	DEh	Ì	254	FEh	■
159	9Fh	ƒ	191	BFh	┐	223	DFh	▀	255	FFh	

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

Floating Point Formats

2.5 Floating-Point Representation

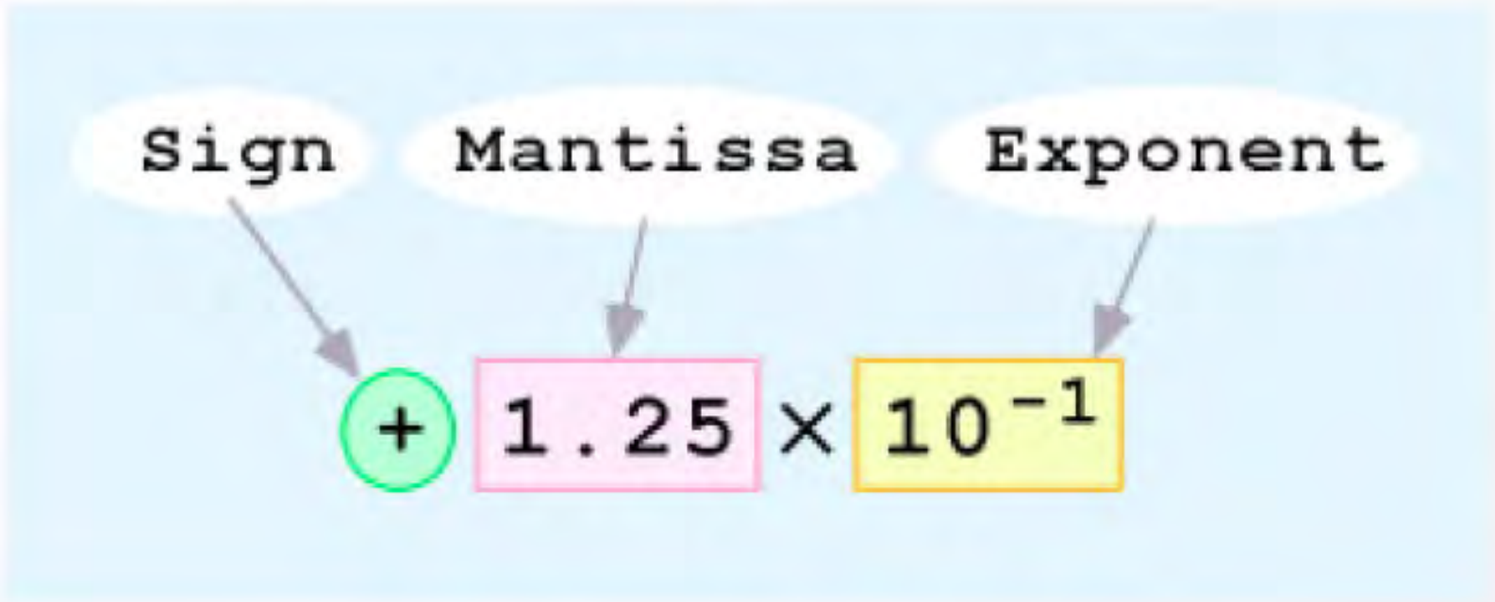
- Floating-point numbers allow an arbitrary number of decimal places to the right of the decimal point.
  - For example: $0.5 \times 0.25 = 0.125$
- They are often expressed in scientific notation.
  - For example:

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

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

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.

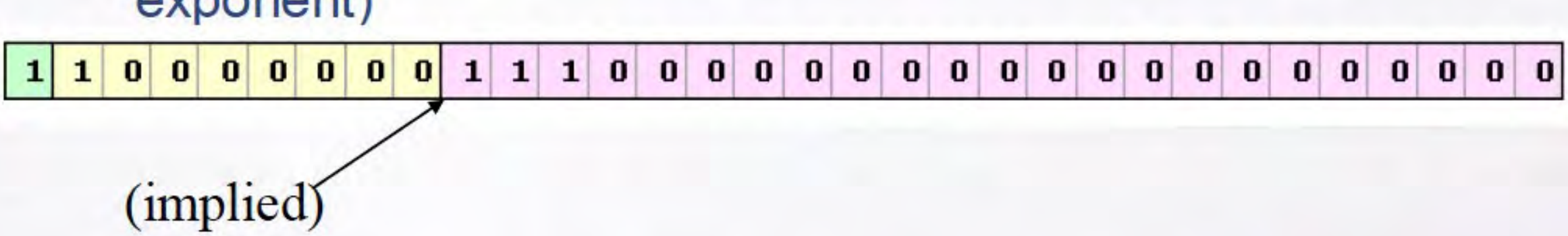
IEEE-754 32-bit Floating Point Format

- 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

UMBC, CMSC313, Richard Chang <chang@umbc.edu>

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:
  - $3.75 = -11.11_2 = -1.111 \times 2^1$
  - 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.