CHAPTER 2
REPRESENTING AND MANIPULATING INFORMATION

CPSC 2310

Dr. Yvon Feaster

Clemson University

2.1 INFORMATION STORAGE

- **Most computers use blocks of bits (8 bits = 1 byte) as the smallest addressable unit of memory**
- **Although our C programs and compilers distinguishes the different data types, the actual machine level program has no knowledge of data types**
- **Everything is simply a block of bytes and the program itself is a sequence of bytes**

IMPORTANT TERMS AND NOTATION

- Below are some terms, with respect to numbers, we will use. To keep confusion down I decided to list these in the beginning of this set of slides. I may add other terms as we progress through this chapter.
  - Decimal - base 10 a.k.a. dec
  - Binary - base 2 a.k.a. bin
    - A binary number may or may not have a “b” at the end or a subscript of 2.
  - Hexadecimal - base 16 a.k.a. hex
    - A number that has 0x or 0X in front of it is a hexadecimal

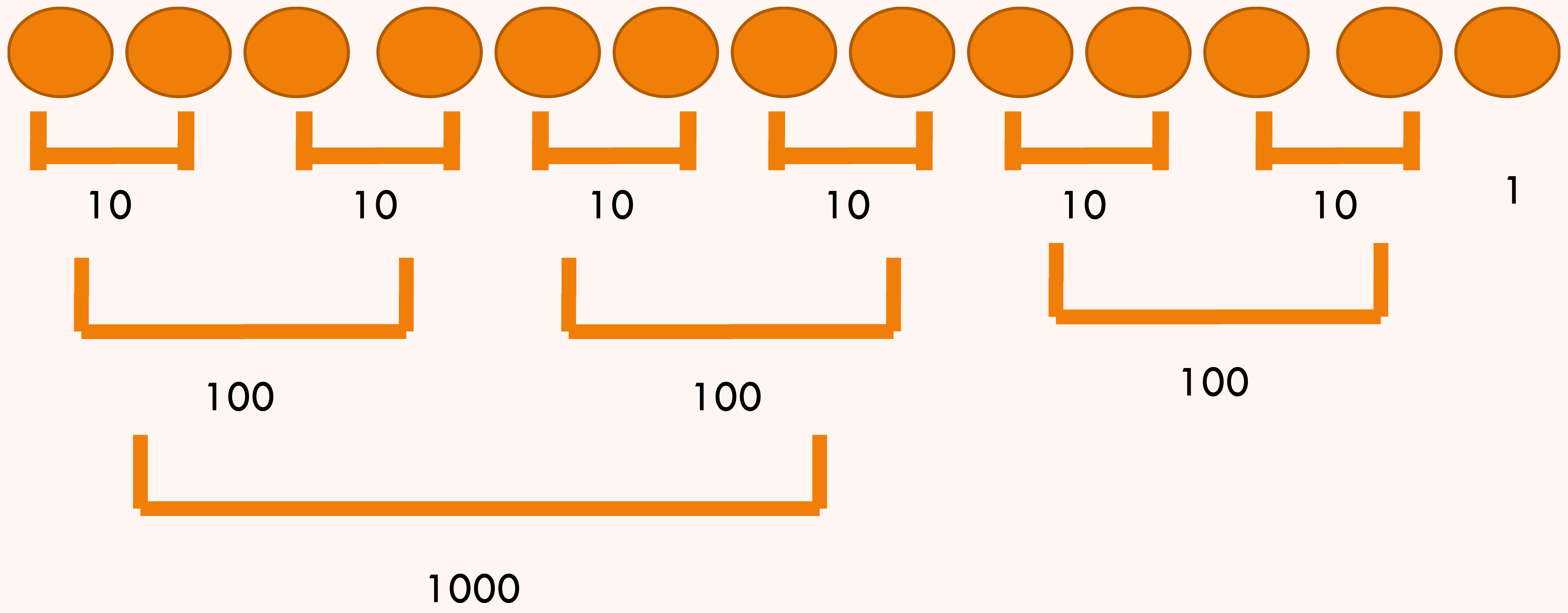
DECIMAL TO BINARY AND VICE VERSA

- Before we go on let's take a quick look at how to convert a decimal to binary
  - Decimal to binary
    - Division
  - Binary to decimal
- These steps can be use with any base

DECIMAL TO BINARY AND VICE VERSA - GROUPING

- **Consider 13 elements (I will draw this out by hand)**

$2_{10} = 10_2$
$4_{10} = 100_2$
$8_{10} = 1000_2$
$16_{10} = 10000_2$



$2_{10} = 10_2$
$4_{10} = 100_2$
$8_{10} = 1000_2$
$16_{10} = 10000_2$

13 =1101

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
				1	1	0	1

(8*1) + (4*1) +(2*0) + (1*1) = 8+4+1 = 13

- **Now lets check this**
- **While this seems kinda interesting it is not very realistic. Suppose I ask you to convert 541 - anyone want to draw out 541 dots???**

DECIMAL TO BINARY AND VICE VERSA

Convert dec 541_{10} to binary

2 | 541	R		
2 | 270	— 1 -	2^0	least significant bit
2 | 135	— 0 -	2^1	
2 | 67	— 1 -	2^2	
2 | 33	— 1 -	2^3	
2 | 16	— 1 -	2^4	
2 | 8	— 0 -	2^5	
2 | 4	— 0 -	2^6	
2 | 2	— 0 -	2^7	
2 | 1	— 0 -	2^8	
2 | 0	— 1 -	2^9	most significant bit

$541_{10} = 1000011101_2$

If you are familiar with the powers of 2.
Another way of converting is subtraction.

541 - 512 = 29 - 16 = 13 - 8 = 5 - 4 = 1

512 + 16 + 8 + 4 + 1 = 541

2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
1024	512	256	128	64	32	16	8	4	2	1
0	1	0	0	0	0	1	1	1	0	1
0	512	0	0	0	0	16	8	4	0	1

DECIMAL TO BINARY

- **Let's try one more:**
- **Decimal 49 to binary**

2.1.1 HEXADECIMAL NOTATION

- **1 byte of unsigned data ranges from 00000000_2 - 11111111_2 with the decimal equivalent of 0 - 255**

- **While a computer actually uses 0's and 1's it is a little verbose, hard for us humans to read. Decimal numbers require a fair amount of work to convert to binary, therefore hexadecimal was chosen (0-9 and A-F) base 16**

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

CONVERTING FROM HEX TO DEC

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

Each hexadecimal number is represented by 4 binary digits

Option 1:
Convert Hex to decimal then multiply by the powers of 16 to get the decimal value

16^4	16^3	16^2	16^1	16^0
65536	4096	256	16	1

How do we convert from Hex_{16} to Dec_{10}?

Hex - C 1 A

First convert hex to decimal

Dec - 12 1 10

$$16^2 + 16^1 + 16^0$$

$$(256*12) + (16*1) + (1*10) = 3098$$

Multiply the decimal number by the equivalent power of 16

CONVERTING FROM HEX TO DEC

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

Option 2:
Convert Hex to binary then multiply by the powers of 2 to get the decimal value

Hex - C 1 A

Bin - 1100 0001 1010

First convert to binary

2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
2048	1024	512	256	128	64	32	16	8	4	2	1
1	1	0	0	0	0	0	1	1	0	1	0
2048	1024	0	0	0	0	0	16	8	0	2	0

$$(1 \times 2^{11}) + (1 \times 2^{10}) + (1 \times 2^4) + (1 \times 2^3) + (1 \times 2^1) = (3098)_{10}$$

$$2048 + 1024 + 16 + 8 + 2 = 3098_{10}$$

Then convert to decimal

EXAMPLES

- **Convert :**
  - **Decimal 183 to binary**
  - **0x4FA6 to binary**
  - **Bin 0110 1001 to dec**
  - **Bin 1110 1001 to hex**

Notice the binary numbers are in two sections of 4 bits

BINARY ADDITION AND SUBTRACTION

- Without converting the entire numbers to decimal we are going to solve the following:
    - 10111 - 111
    - 11110 + 01110 + 00111
    - 1000 - 1

HEX ADDITION AND SUBTRACTION

- Without converting the entire numbers to decimal we are going to solve the following:
  - 0x503C + 0x8
  - 0x503C -0x40
- Conversion websites

2.1.1 CONTINUED

- When a value ‘x’ is a power of 2, $x = 2^n$ for some nonnegative integer ’n’, we can write ‘x’ in hexadecimal form by realizing that the binary representation of ‘x’ is simply 1 followed by ’n’ zeros.
  - Ex. $x = 64 = 2^6$ = 1000000 = 0100 0000 = 0x40
  - Another way to look at this:
    - Given $64 = 2^6$ - 6 being the superscript (power) so divide by 4 (6/4 = 1 with a remainder of 2).
    - For every group of 4 zeros in binary this represents one 0 in hex. The remainder represents the extra 0’s with a 1 in front so we have a 1 and 2 zeros = 100 in binary = 4 in hex plus the 4 extra 0’s = a single 0 in hex - therefore we have 0x40.

2.1.1 CONTINUED

- What if given
  - 0x100 = 2^n , What is n?
- What do we know about each zero in 0x100?
- What is ‘n’ if we have 0x4000 is 2^n

2.1.1 CONTINUED

- **As practice we will fill in the entries in the following table**

n	2^n Hexadecimal
9	0x200
19	
	0x10000
17	
	0x80

2.1.2 DATA SIZES

- All computers have a word size. In most modern computers that is 8 bytes = 64-bits
- The word size indicates the nominal size of a pointer. Let's see what the size of a pointer is on my laptop.
  - CODE: wordSize.c
- Most 64 bit machines will run programs compiled on a 32 bit machine
- However, 32 bit machines will not run programs compiled for 64 bit machines

2.1.2 DATA SIZES

- **The C programming language supports multiple data formats for both integer and floating point data**
- **The sizes of some of these depend on the computer's word size**

If you want a particular
Size then specify

dataSizes.c

C Declaration		Bytes per machine	
Signed	**Unsigned**	**32-bits**	**64-bits**
char	unsigned char	1	1
short	unsigned short	2	2
int	unsigned	4	4
long	unsigned long	4	8
int32_t	uint32_t	4	4
int64_t	uint64_t	8	8
char*		4	8
float		4	4
double		8	8

2.1.3 ADDRESSING AND BYTE ORDERING

- When data spans multiple bytes two things must be decided
  - First: What the address of the object will be
    - Multi-byte data is most always stored as a contiguous sequence of **bytes**
    - Ex. Suppose variable int x has an address of 0x100 - since an int is a 4 byte piece of data the addresses would be:

      0x100, 0x101, 0x102, 0x103

      - CODE: intArray.c

Assume arr is array of integers

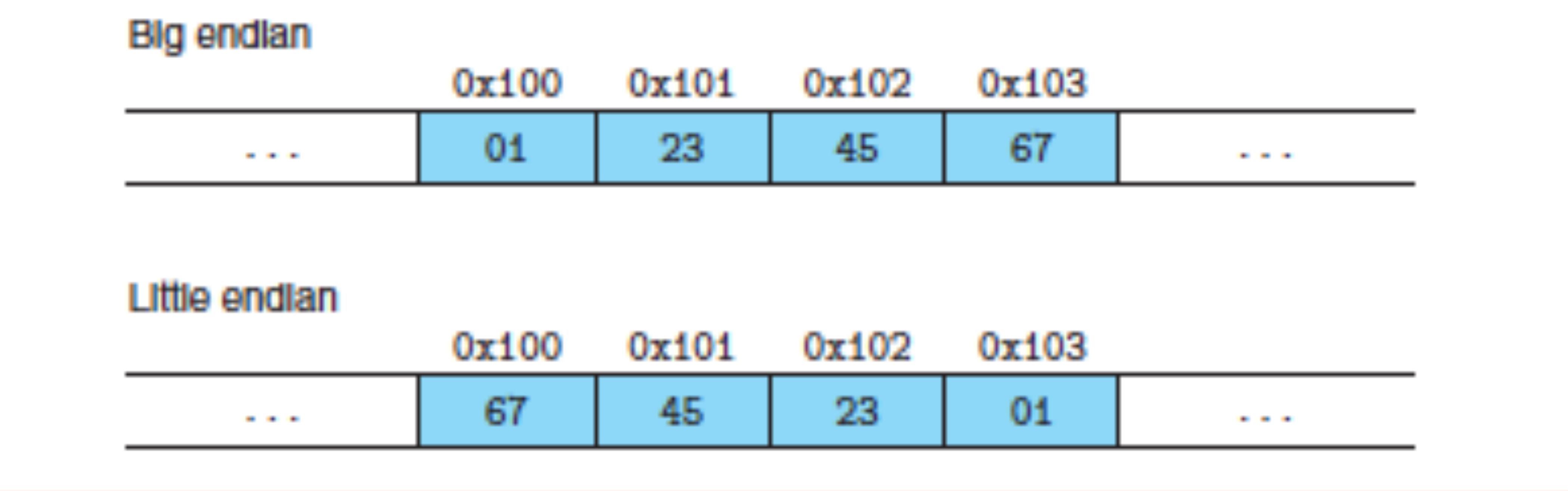
address of arr[0]: 7FFEE71258B0
address of arr[1]: 7FFEE71258B4
address of arr[2]: 7FFEE71258B8
address of arr[3]: 7FFEE71258BC
address of arr[4]: 7FFEE71258C0

address of dArr[0]: 7FFEE7125880
address of dArr[1]: 7FFEE7125888
address of dArr[2]: 7FFEE7125890
address of dArr[3]: 7FFEE7125898
address of dArr[4]: 7FFEE71258A0

2.1.3 ADDRESSING AND BYTE ORDERING

- When data spans multiple bytes two things must be decided
  - Second: How will the bytes be ordered in memory
  - 2 ways bytes are ordered - this is determined by the computer you are using
    - Big endian - most significant byte to the least significant byte
    - Little endian - least significant byte to the most significant byte

Assume some data has the value of 0x01234567

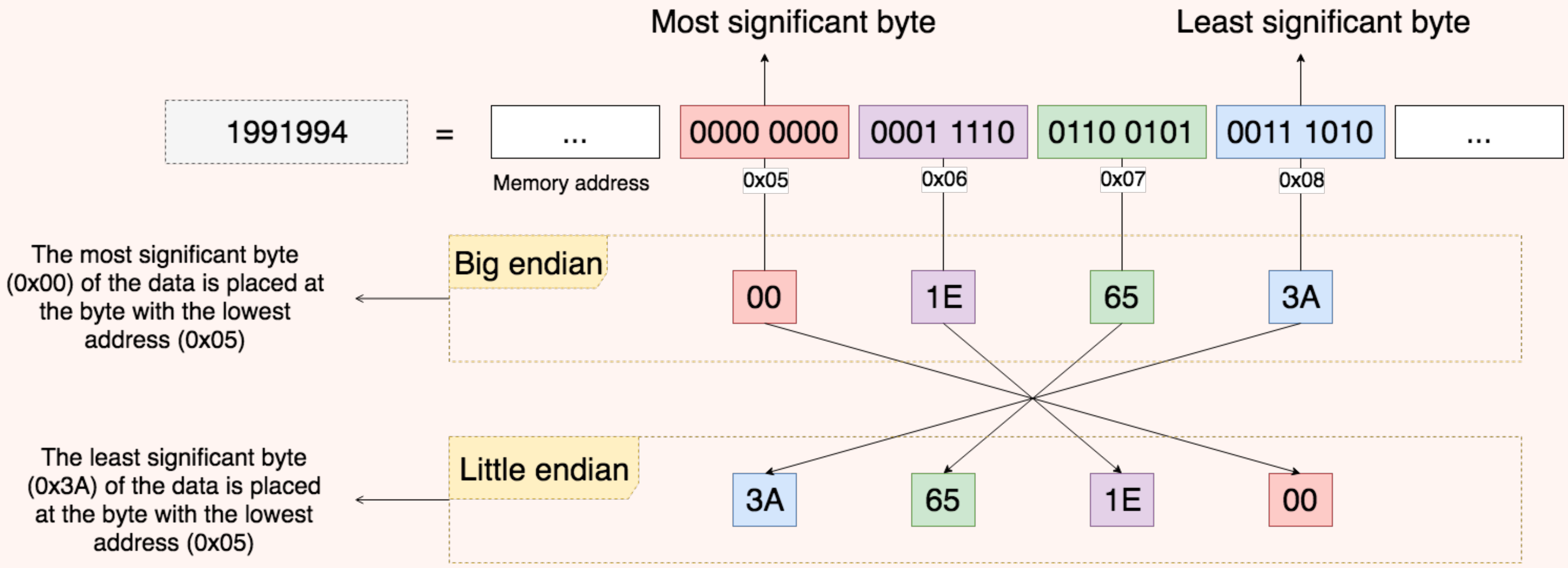


The value of a signed int's range is -2147483648 to 2147483647

Suppose the value of an int is 2147483647 in hex this is 0x7FFFFFFF which is 4 groups of 8 bits 7F FF FF FF

Depending on the machine you are running these could possibly be stored as: FFFFFF7F (Little Endian) or 7FFFFFFF (Big Endian)

BIG AND LITTLE ENDIAN



2.1.3. ADDRESSING AND BYTE ORDERING

12345 in decimal is 00003039 (4 bytes) in hex

Machine	Value	Type	Bytes(hex)
Linux 32	12345	int	39 30 00 00
Windows	12345	int	39 30 00 00
Sun	12345	int	00 00 30 39
Linux 64	12345	int	39 30 00 00

Big Endian - most significant byte to least significant byte
Little Endian - least significant byte to most significant byte

FUN FACT

- The origins of the terms Big Endian and Little Endian was a dispute that revolved around the proper way to break a boiled egg.
    - Derived from Jonathan Swift’s Gulliver Travels
        - Lilliputian King required all of subjects “Little Endians” to break their eggs on the small end of the egg.
        - A political faction called the “Big Endians” rebelled and broke their eggs on the large end of the egg.

2.1.3 ADDRESSING AND BYTE ORDERING

- **Some processors are big endian, some are little endian there are also bi-endian which means you can choose which flavor you want.**
- **However, byte order is fixed once an operating system is chosen.**
  - **Example: ARM microprocessors, used in many cellphones, have hardware that can operate in either little or big endian but the two most common operating systems for those chips Android and IOS operate in little-endian mode**
- **The reason both big and little endian coexist is, in early days, the different CPU makers used different conventions for representing multibyte data, and no standard emerged at the time.**
- **For most application programming, byte ordering is invisible and a program ran on either class of machines give identical results**
- **However!!!!**

2.1.3 ADDRESSING AND BYTE ORDERING

- There are at least 3 instances when it matter
  - Network communication: When binary data will be communicated over a network between different machines. If one machine is little endian and the other is big endian then there will be problems with translating the data. This requires that code written for networking applications must follow established conventions for byte ordering to make sure the sending machine converts its internal representation to the network standards
  - When representing an address in the assembly language
    - Ex. A disassembler may show (Hex byte sequence in memory) | 430b2000 | - little endian
    - an assembly instruction shows them in big endian style this is confusing
  - Could become a problem when we type cast variables in C (circumvents normal type system and can cause portability problems)

2.1.3 ADDRESSING AND BYTE ORDERING

- Let's run a quick program that will print the hex value of:
  - 12345 is represented in hex by 0x3039
  - 0x7FFFFFFF is hex for 2147483647 (a signed int has the value of -2147483647 to 2147483647)
  - The character string of "fedcba"
- A couple things to remember about C strings:
  - They are encoded by an array of characters terminated by null (0)
  - ASCII is the standard scheme for characters
- What endian is this little or Big?
  - show_bytes1.c

2.1.6 INTRODUCTION TO BOOLEAN ALGEBRA

- There is a great deal of mathematical knowledge that revolves around the study of the values of 0 and 1. Much of this research started with George Boole around 1850, hence Boolean algebra
- Boolean algebra encodes logic values of True/False
- Logical operations include NOT, AND, OR, EXCLUSIVE-OR
- We can extend this concept to work with strings of 1's and 0's

~	0	1
	1	0

Not

&	0	1
0	0	0
1	0	1

And

|	0	1
0	0	1
1	1	1

Or

^	0	1
0	0	1
1	1	0

Exclusive-Or

BITWISE ~(NOT)

- **~ (NOT) - Unary operator that flips the bits of the number if a bit is 1 then it becomes 0 and if a bit is 0 it becomes 1**
  - **Example:**
    - **N = 5 = 101_2**
    - **~N = ~5 = 010_2 = 2**

$$\begin{array}{cccc} \sim & 1 & 0 & 1 \\ \hline & 0 & 1 & 0 \end{array}$$

BITWISE AND(&)

&	0	1
0	0	0
1	0	1

- **& - AND is a binary operator that operates on two equal-length bit patterns. If both bits are 1, the resulting bit will be 1, otherwise 0.**
  - **Example:**
    - **A = 5 = $(101)_2$, B = 3 $(011)_2$ suppose A&B = $(101)_2$ & $(011)_2$ = $(001)_2$ = 1**

$$
\begin{array}{cccc}
 & 1 & 0 & 1 \\
\& & 0 & 1 & 1 \\
\hline
 & 0 & 0 & 1
\end{array}
$$

BITWISE OR (|)

- | - OR is a binary operator that operates on two equal-length bit patterns, similar to bitwise &. If both bits are 0, the resulting bit is 0, otherwise it is 1. Or you could say **if either bit is 1 the result is 1.**
  - Example
    - A = 5 = $(101)_2$, B = 3 $(011)_2$ suppose A|B = $(101)_2$ | $(011)_2$ = $(111)_2$ = 7

$$
\begin{array}{cccc}
 & 1 & 0 & 1 \\
| & 0 & 1 & 1 \\
\hline
 & 1 & 1 & 1
\end{array}
$$

BITWISE ^ (XOR)

- ^ - Takes two equal-length bit patterns. If both bits are the same, either two 0's or two 1's the result is 0. Otherwise the result is 1. (If both are the same you get a 0. If different you get a 1.)
  - Example
    - A = 5 = $(101)_2$, B = 3 $(011)_2$ suppose A^B = $(101)_2$ ^ $(011)_2$ = $(110)_2$ = 6

Inputs		Output
A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

$$
\begin{array}{rccc}
 & 1 & 0 & 1 \\
\wedge & 0 & 1 & 1 \\
\hline
 & 1 & 1 & 0
\end{array}
$$

ANOTHER EXAMPLE

- Consider the example where W = 4 (word size) with arguments of a = [0110] and b = [1100], then the operations a & b,   a | b,   a ^ b, and ~b should yield what. Let's work through these together. **bitWise.c**

2.1.7 BIT-LEVEL OPERATION IN C

- A common use of bit-level operations is to implement masking operations, where a mask is a bit pattern that indicates a selected set of bits within a word.
- A mask defines which bits you want to keep, and which bits you want to clear
- A mask can also be use to toggle or set a single bit (flags)
- Masking is the act of applying a mask to a value. Examples:
  - Bitwise ANDing in order to extract a subset of the bits in the value
  - Bitwise ORing in order to set a subset of the bits in the value
  - Bitwise XORing in order to toggle a subset of the bits in the values

2.1.7 BIT-LEVEL OPERATION IN C

Ex:
X = 0x89ABCDEF = 4 bytes = 32 bits
Mask = 0xFF = we are masking 4 bytes so it is translated to 0x000000FF
This example will return the last byte of X

&	0	1
0	0	0
1	0	1

		1000	1001	1010	1011	1100	1101	1110	1111	
Mask	&	0000	0000	0000	0000	0000	0000	1111	1111	
		0000	0000	0000	0000	0000	0000	1110	1111	= 0x000000EF

2.1.7 OTHER MASK

If I want to know all <u>but</u> the least significant byte I could use X & 0xFFFFFF00 (this would only work with 32 bits.) What if I want this to work with all words sizes. X &(~0xFF)

&	0	1
0	0	0
1	0	1

X in binary

0x87654321		1000 0111 0110 0101 0100 0011 0010 0001
~0xFF	&	1111 1111 1111 1111 1111 1111 0000 0000
		1000 0111 0110 0101 0100 0011 0000 0000

2.1.7 OTHER MASK

If I want all <u>but</u> the least significant byte of X complemented, I would use X^(~0xFF)

(Complements everything but the last byte.)

^	0	1
0	0	1
1	1	0

X in binary

0x87654321

		1000	0111	0110	0101	0100	0011	0010	0001
~0xFF	^	1111	1111	1111	1111	1111	1111	0000	0000
		0111	1000	1001	1010	1011	1100	0010	0001
		7	8	9	A	B	C	2	1

2.1.7 OTHER MASK

If I want to set the least significant byte to be all 1's but not change any of the other bytes I would use X | 0xFF

|	0	1
0	0	1
1	1	1

X in binary

0x87654321		1000	0111	0110	0101	0100	0011	0010	0001
0xFF	|	0000	0000	0000	0000	0000	0000	1111	1111
		1000	0111	0110	0101	0100	0011	1111	1111

2.1.7 COUPLE PROPERTIES OF BIT-LEVEL OPERATIONS

- **Yvon's aside:**

I picked up a couple interesting properties with respect to bit level operations

X ^ Y = (X & ~Y) | (~X & Y)

Sum of Products

X ^ 0s = X	X & 0s = 0	X | 0s = X
X ^ 1s = ~X	X & 1s = X	X | 1s = 1s
X ^ X = 0	X & X = X	X | X = X

Determining if a number is odd is another common use of the bit operator &

You should work through these and not just memorize them. Understanding these properties could be helpful.

LEFTSHIFT (<<)

- << - Left shift operator is a binary operator which shifts N number of bits to the left and appends N 0's to the end (right).
- x << k means x is shifted k bits to the left, dropping off the k most significant bits and filling the right end with k zero's
- Shift operations associate from left to right, so x << j << k is equivalent to (x << j) << k.
- Left shift is the equivalent to multiplying the bit pattern with 2^k (assuming shift k bits)

Assumes 1 byte:

0000 0001 << k = 00000001 * 2^k
0000 0001 << 1 = 0000 0010 = 2 = 2^1
0000 0001 << 2 = 0000 0100 = 4 = 2^2
0000 0001 << 3 = 0000 1000= 8 = 2^3
0000 0001 << 4 = 0001 0000 = 16 = 2^4

0000 0011 << 1 = 0000 0110 = ($3*2^1$) = (3*2) = 6
0000 0011 << 2 = 0000 1100 = ($3*2^2$) =(3*4) = 12
0000 0011 << 3 = 0001 1000 = ($3*2^3$) =(3*8) = 24
0000 0011 << 4 = 0011 0000 = ($3*2^4$) =(3*16) = 48
Etc.

RIGHT SHIFT (>>)

- **>> is a binary operator which shifts the same number of bits, in the given bit pattern, to the right and appends to the left.**
- **Right shift can be used to divide by 2^k where k is the number of bits you are shifting**

Assumes 1 byte:
0001 0000 >> k where k is 2^k
0001 0000 >> 1 = 0000 1000 = 8 = $16/2^1$
0001 0000 >> 2 = 0000 0100 = 4 = $16/2^2$
0001 0000 >> 3 = 0000 0010= 2 = $16/2^3$
0001 0000 >> 4 = 0000 0001 = 1 = $16/2^4$
What about:
0000 0101 >> 1 = 0000 0010 = 2 =$5/2^1$ = 2.5 = 2
.5 gets truncated

BIT SHIFT OPERATIONS IN C

- I have already mentioned that shift operation is associative from left to right, so x << j << k is equivalent to (x << j) << k.
- You should also pay attention to operator precedence issues with shift operations
- Ex. 1 << 2 + 3 << 4 may be intended to be (1 << 2) + (3 << 4), however addition and subtraction has a high precedence. So it would add 2 and 3 then apply the shift operator
- Make sure you use ( ) to indicate the order of operation. When in doubt use ( ).

RIGHT SHIFT >>

- Most architectures support 2 forms of right shift
  - Logical
  - Arithmetic

OPERATION	VALUE 1	VALUE 2
ARGUMENT X	01100011	10010101
X<<4	00110000	01010000
X>>4 logically	00000110	00001001
X>>4 arithmetic	00000110	11111001

- On the arithmetic example, value 2's most significant bit is 1 so it fills in with 1's to preserve the sign of signed integers (we will discuss signed/unsigned later)
- "C" does not specify that 1 will be used w.r.t. arithmetic, but this could cause probability problems so most compilers/computers use arithmetic right shifts on signed data
- Unsigned uses logical right shift.

X in Hex	X in Binary	X<< 3	X>>2 (logical)	X>>2 (arithmetic)
0xC3	1100 0011	0001 1000	0011 0000	1111 0000
0x75				
0x87				
0x66	0110 0110	0011 0000	0001 1001	0001 1001

2.1.8 LOGICAL OPERATIONS IN C

- You should already be aware of the logical operators provided by "C"
  - && - and
  - || - or
  - ! - not
- While these seem to be the same &, |, ~ they have different behavior
- The logical operators evaluate to :

  TRUE = 1 = 0x01 or something other that 0

  FALSE = 0 = 0x00

Expression	Result
!0x41 => !(True)	0x00 False
!0x00 => !(False)	0x01 True
!!0x41 => !(!(0x41)) !(!0x41) => !(False)	0x01 True
0x69 && 0x55 True && True	0x01 True
0x69 || 0x55 True || True	0x01 True

2.2 INTEGER REPRESENTATIONS (SIGNED VS UNSIGNED)

- To this point we have only worked with unsigned numbers with respect to converting to binary.
  - Unsigned integers represent 0 and positive integers
- Next we will take a peek at how the system represents signed integers.
  - Signed integer represent 0, positive, and negative integers
  - When dealing with signed numbers the range of a number changes because one bit is used as the signed bit.

2.2 INTEGER REPRESENTATIONS (UNSIGNED)

- In this section we will look at the value ranges for unsigned and signed
- We will start with unsigned which is what we have dealt with so far.
- Let's consider a char which is 1 byte (8 bits) in size
  - A char can be represented by integers in the range of 0 - 255.
  - How do we know this? What is the value of the binary number 1111 1111?

The data range for an unsigned char:
8 bits (0 to (2^8 - 1))
$0000\ 0000_2$ to $1111\ 1111_2$ or
0_{10} to 255_{10} or
0x00 to 0xFF

So we can say that (0 to (2^w - 1)) is the range for any integer data type with 'w' being the number of bits.
What would 'w' be for an int?

2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
256	128	64	32	16	8	4	2	1
	1	1	1	1	1	1	1	1

2.2 INTEGER REPRESENTATIONS (SIGNED)

- The range for signed data is (-2^{w-1}) to $(+2^{w-1} - 1)$
- Consider int8_t a.k.a. signed int with 8 bits. However the most significant bit represents the sign of the number (negative/positive).
- This means int8_t (w = 8) but since it is signed it only has 7 bits to represent the data. (One less than unsigned)
- (-2^{w-1}) to $(+2^{w-1} - 1)$
  - -2^7 to $2^7 - 1$ = -128 to 127

2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
256	128	64	32	16	8	4	2	1

CALCULATING VALUES FOR SIGNED AND UNSIGNED

- **What is the value of 10101010_2 - unsigned**

  128 + 32 + 8 + 2 = 170

- **What is the value of 10101010_2 - signed**

  MSB = 1 so this is a negative number

  -128 + 32 + 8 + 2 = -86

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
1	0	1	0	1	0	1	0

TYPICAL RANGES FOR C INTEGRAL DATA TYPES FOR 32 AND 64 BIT PROGRAMS

32 bits

C data type	Minimum	Maximum
[signed] char	−128	127
unsigned char	0	255
short	−32,768	32,767
unsigned short	0	65,535
int	−2,147,483,648	2,147,483,647
unsigned	0	4,294,967,295
long	−2,147,483,648	2,147,483,647
unsigned long	0	4,294,967,295
int32_t	−2,147,483,648	2,147,483,647
uint32_t	0	4,294,967,295
int64_t	−9,223,372,036,854,775,808	9,223,372,036,854,775,807
uint64_t	0	18,446,744,073,709,551,615

Figure 2.9 Typical ranges for C integral data types for 32-bit programs.

64 bits

C data type	Minimum	Maximum
[signed] char	−128	127
unsigned char	0	255
short	−32,768	32,767
unsigned short	0	65,535
int	−2,147,483,648	2,147,483,647
unsigned	0	4,294,967,295
long	−9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long	0	18,446,744,073,709,551,615
int32_t	−2,147,483,648	2,147,483,647
uint32_t	0	4,294,967,295
int64_t	−9,223,372,036,854,775,808	9,223,372,036,854,775,807
uint64_t	0	18,446,744,073,709,551,615

Figure 2.10 Typical ranges for C integral data types for 64-bit programs.

2.2 INTEGER REPRESENTATIONS (SIGNED VS UNSIGNED)

- **Three schemes used to encode integers were developed over the years. Two of which have flaws.**
  - **Sign-Magnitude**
  - **1's Complement**

Both of these have two ways to represent the value of 0

  - **2's complement**

This one only has one way to represent the value 0 and is the ideal version of how to represent negative numbers. 2's complement is the scheme most used for integer representation and the scheme we will concentrate on in this class.

2.2 INTEGER REPRESENTATIONS (SIGNED VS UNSIGNED)

-/+	4	2	1		
1	1	1	1	=	-7
1	1	1	0	=	-6
1	1	0	1	=	-5
1	1	0	0	=	-4
1	0	1	1	=	-3
1	0	1	0	=	-2
1	0	0	1	=	-1
1	0	0	0	=	-0
0	0	0	0	=	0
0	0	0	1	=	1
0	0	1	0	=	2
0	0	1	1	=	3
0	1	0	0	=	4
0	1	0	1	=	5
0	1	1	0	=	6
0	1	1	1	=	7

Two ways to Represent 0

- **Signed-magnitude - the value of the integer bits added together except the most significant bit.**
  - **If the value is negative the most significant bit will be 1**
    - **Ex. +53 = 0011 0101 and -53 = 1011 0101**

$$\begin{array}{r} 5 \\ \underline{+(-5)} \\ ?? \end{array} \qquad \begin{array}{r} 0101 \\ \underline{1101} \\ \cancel{1}0010 \end{array} = 2$$

This is a problem

2.2 INTEGER REPRESENTATIONS (SIGNED VS UNSIGNED)

-/+	4	2	1		
1	0	0	0	=	-7
1	0	0	1	=	-6
1	0	1	0	=	-5
1	0	1	1	=	-4
1	1	0	0	=	-3
1	1	0	1	=	-2
1	1	1	0	=	-1
1	1	1	1	=	-0
0	0	0	0	=	0
0	0	0	1	=	1
0	0	1	0	=	2
0	0	1	1	=	3
0	1	0	0	=	4
0	1	0	1	=	5
0	1	1	0	=	6
0	1	1	1	=	7

Two ways to represent 0

- **For 1's compliment, if the most significant bit (MSB) is 1 then the number is negative**

To determine the value of a 1's compliment number

- **Positive - the MSB will be 0. Simply add the values.**
- **If Negative - the MSB will be 1. To determine the value, complement all but the most significant bit.**
  - **Ex. For 1's complement:**

What is the value of 1000 0001 = 1111 1110 = 64 + 32 +16 + 8 + 4 + 2 = 126 - since MSB is 1 then this is -126

5	0101	
+(-5)	1010	
??	1111	= -0

This is better than signed magnitude. But still has problems. Lets take a look at a couple other examples. 5 + (-3) or 6 +(-2)

REVIEW THE RULES

- If given the binary:
  - Signed-magnitude
    - To get the value - add all bits except most significant.
    - If the value is negative the MSB will be 1, positive MSB will be 0
  - 1's compliment
    - To get the Values:
    - If the MSB is 0 simply add the values.
    - If Negative - the MSB will be 1. To determine the value, complement all but the most significant bit.
- With the above information you should be able to convert from decimal to binary for each type

EXAMPLE

Sign-Magnitude convert the following:

35 = 00100011

-35 = 10100011

0111 0010 = msb = positive

0+64+32+16+2 = 114

1111 0010 = msb = negative

64+32+16+2 = -114

1's complement convert the following:

35 = 00100011

-35 = find positive then flip the bits

00100011 = 11011100

0111 0010 = msb = 0 = add bits

0+64+32+16+2

10101010 = Flip all but msb - then add

11010101 = 64 + 16 + 4 + 1 = 85 msb = 1 so -85

2'S COMPLEMENT

-/+	4	2	1		
1	0	0	0	=	-8
1	0	0	1	=	-7
1	0	1	0	=	-6
1	0	1	1	=	-5
1	1	0	0	=	-4
1	1	0	1	=	-3
1	1	1	0	=	-2
1	1	1	1	=	-1
0	0	0	0	=	0
0	0	0	1	=	1
0	0	1	0	=	2
0	0	1	1	=	3
0	1	0	0	=	4
0	1	0	1	=	5
0	1	1	0	=	6
0	1	1	1	=	7

Only one way to
Represent 0

$$\begin{array}{r} 5 \\ \underline{+(-5)} \\ ?? \end{array} \qquad \begin{array}{r} 0101 \\ \underline{1011} \\ \cancel{1}0000 \end{array}$$

- **Since both sign-magnitude and 1's complement have problems**
- **Most systems use 2's complement to encode signed data. So what is 2's complement?**
  - **If given 127 = 0111 1111, we can determine -127 using 2's complement.**
  - **The rule for 2's complement is complement the bits then add 1.**
  - **So ~(0111 1111) =1000 0000 + 1 = 1000 0001 = -128 + 1 = -127 (since 1 is in the most significant bit position then 128 is -128)**

~(01111 1111) + 1 =

$$\begin{array}{lr} & 1000\ 0000 \\ + & \underline{\qquad\qquad 1} \\ & 1000\ 0001 = (-128 + 1) = -127 \end{array}$$

2'S COMPLEMENT

- So what if I have a negative number and I want to determine the positive value? Use 2's complement on that.
- Suppose I have -97 which is 1001 1111 and I want to know what 97 is in binary. Take the 2's complement of it.

1001 1111 = -97
2's complement
    0110 0000-flip all the bits
  + <u>   1</u>
    0110 0001 = 64 + 32 + 1 = 97

2'S COMPLEMENT TRICK

- If given a binary number and you want the two's complement, use what I call the Scan Method.
- Scan right to left flipping the bits after the first 1 then calculate the value.

We will look at 97 since we already know what the + and - values are.

Given 0110 0001 = 97 scan right to left preserving the bits until after the first 1, then flip the remaining bits.
1001 1111

Another Example:

0110 1100 = 108 - find -108

1010 1100 = -84 - find 84

REVIEW THE RULES

- Signed-magnitude
  - To get the value add all bits except most significant.
  - If the value is negative the MSB will be 1, positive MSB will be 0
- 1's compliment
  - To get the Values:
    - If the MSB is 0 simply add the values.
    - If Negative - the MSB will be 1. To determine the value, complement all but the most significant bit.
- 2's compliment
  - To get value:
    - If MSB is 1 - value is negative, if 0 value is positive
    - If given + complement the bits then add 1 to get negative
    - If given - complement the bits then add 1 to get positive

CONVERT PRACTICE

- **If given a number in sign-magnitude, 1's complement, or 2's complement you should be able to determine the binary representation of the remaining two.**

Int	Signed Magnitude	1's Complement	2's complement
			1010 1100
49			
		1100 1010	
-75			
	0110 0110		

CONVERT PRACTICE

- **If given a number in sign-magnitude, 1's complement, or 2's complement you should be able to determine the binary representation of the remaining two.**

Int	Signed Magnitude	1's Complement	2's complement
-84	1101 0100	1010 1011	1010 1100
49	0011 0001	0011 0001	0011 0001
-53	1011 0101	1100 1010	1100 1011
-75	1100 1011	1011 0100	1011 0101
102	0110 0110	0110 0110	0110 0110

2.2.3 TWO'S COMPLEMENT ENCODING

- FROM THIS POINT ON WE WILL USE 2'S COMPLEMENT. THIS IS THE ENCODING SCHEME USED BY MOST COMPUTERS
- On the next slide we will see a chart that shows the MAX and MIN values for different word sizes. The C programming language has created various constants that represent the MIN and MAX of different data types. These can be found in the limits.h library.
- As an example: CHAR_MAX, CHAR_MIN, INT_MAX, INT_MIN,etc.
- The limits.h has signed and unsigned representation for short, long, char and int

C CONSTANTS FOR IMPORTANT NUMBERS

Unsigned Values

UMin = 0

UMax = $2^w - 1$

Value	Word size w: 8	16	32	64
$UMax_w$	0xFF	0xFFFF	0xFFFFFFFF	0xFFFFFFFFFFFFFFFF
	255	65,535	4,294,967,295	18,446,744,073,709,551,615
$TMin_w$	0x80	0x8000	0x80000000	0x8000000000000000
	−128	−32,768	−2,147,483,648	−9,223,372,036,854,775,808
$TMax_w$	0x7F	0x7FFF	0x7FFFFFFF	0x7FFFFFFFFFFFFFFF
	127	32,767	2,147,483,647	9,223,372,036,854,775,807
−1	0xFF	0xFFFF	0xFFFFFFFF	0xFFFFFFFFFFFFFFFF
0	0x00	0x0000	0x00000000	0x0000000000000000

Take note of this line.

Figure 2.14 Important numbers. Both numeric values and hexadecimal representations are shown.

2's Complement Values

TMin = -2^{w-1}

TMax = $2^{w-1} - 1$

2.2.3 TWO'S COMPLEMENT ENCODING

Signed

Unsigned

	12,345		−12,345		53,191	
Weight	Bit	Value	Bit	Value	Bit	Value
1	1	1	1	1	1	1
2	0	0	1	2	1	2
4	0	0	1	4	1	4
8	1	8	0	0	0	0
16	1	16	0	0	0	0
32	1	32	0	0	0	0
64	0	0	1	64	1	64
128	0	0	1	128	1	128
256	0	0	1	256	1	256
512	0	0	1	512	1	512
1,024	0	0	1	1,024	1	1,024
2,048	0	0	1	2,048	1	2,048
4,096	1	4,096	0	0	0	0
8,192	1	8,192	0	0	0	0
16,384	0	0	1	16,384	1	16,384
±32,768	0	0	1	−32,768	1	32,768
Total		12,345		−12,345		53,191

Figure 2.15 Two's-complement representations of 12,345 and −12,345, and unsigned representation of 53,191. Note that the latter two have identical bit representations.

What do you notice about these circled binary numbers? What distinguishes these two values?

2.2.5 SIGNED VS UNSIGNED IN C

- **The C Language allows for the casting of signed and unsigned numbers.**

- **Casting can be explicit:**
  int tx, ty;
  unsigned int ux, uy;
  tx = (int) ux;
  uy = (unsigned) ty;

- **Or Casting can be implicit:**
  int tx, ty;
  unsigned ux, uy;
  tx = ux;
  uy = ty;

- **Also, we can see that we can cast when printing using printf**

“int” is implied

int x = -1;
unsigned u = 2147483648; this is (2^{31})

printf(“x=%u=%d\n”, x,x)
printf(“u=%u=%d\n”, u,u)

u prints unsigned decimal and d prints signed decimal

Here is what will print for a 32 and 64 bit machine
x = 4294967295 = -1
u = 2147483648 = -2147483648

$UMax_w$	0xFF	0xFFFF	0xFFFFFFFF	0xFFFFFFFFFFFFFFFF
	255	65,535	4,294,967,295	18,446,744,073,709,551,615
$TMin_w$	0x80	0x8000	0x80000000	0x8000000000000000
	−128	−32,768	−2,147,483,648	−9,223,372,036,854,775,808
$TMax_w$	0x7F	0x7FFF	0x7FFFFFFF	0x7FFFFFFFFFFFFFFF
	127	32,767	2,147,483,647	9,223,372,036,854,775,807

2.2.6 EXPANDING THE BIT REPRESENTATION OF AN NUMBER

- "One common operation is to convert between integers having different sizes while retaining the same numeric value. Of course, this may not be possible when the destination data type is too small to represent the desired value. Converting from a smaller to larger data type should always be possible." Course Book
- Principle: Expansion of an unsigned number by ZERO EXTENSION
- Principle: Expansion of a signed two's-complement number by SIGNED EXTENSION

- Zero Extension Example

Given uint8_t a = 0b1000 0000;

uint16_t b = a;

b would have the value of

0000 0000 1000 000

- Signed Extension Example

Give int8_t a = 0b1000 0000;

int16_t b = a;

b would have the value of

1111 1111 1000 0000

EXPANDING THE BIT REPRESENTATION OF A NUMBER PRACTICE

- Prove to yourself that the following bit vectors is a two's complement representation of -5.

    1. [1011]

    2. [11011]

    3. [111011]

- It does not matter how many 1's you extend this number by you will still get -5

expansion.c

We will now look at what happens
When we make change data types.

Two things to consider:
1. Is there a size change
2. Is there a change in sign

Always consider the right-hand side first.

Principle: Expansion of an unsigned number by ZERO EXTENSION

Assignment	Method
char = unsigned char	Preserve bit pattern; high-order bit becomes sign bit
short = unsigned char	Zero-extend
long = unsigned char	Zero-extend
unsigned short = unsigned char	Zero-extend
unsigned long = unsigned char	Zero-extend
char = unsigned short	Preserve the low-order byte
short = unsigned short	Preserve bit pattern; high-order bit becomes sign bit
long = unsigned short	Zero-extend
unsigned char = unsigned short	Preserve low-order byte
char = unsigned long	Preserve low-order byte
short = unsigned long	Preserve low-order byte
long = unsigned long	Preserve bit pattern; high-order bit becomes sign bit
unsigned char = unsigned long	Preserve low-order byte
unsigned short = unsigned long	Preserve low-order byte

We will now look at what happens
When we make change data types.

Two things to consider:

1. Is there a size change
2. Is there a change in sign

Always consider the right-hand side first.

Principle: Expansion of a two’s-complement number by SIGNED EXTENSION

Assignment	Method
short = char	Sign-extend
long = char	Sign-extend
unsigned char = char	Preserve pattern; high-order bit loses function as sign bit
unsigned short = char	Sign-extend char to short; convert short to unsigned short
unsigned long = char	Sign-extend char to long; convert long to unsigned long
char = short	Preserve low-order byte
long = short	Sign-extend
unsigned char = short	Preserve low-order byte
unsigned short = short	Preserve pattern; high-order bit loses function as sign bit
unsigned long = short	Sign-extand short to long; convert long to unsigned long
char = long	Preserve low-order byte
short = long	Preserve low-order bytes
unsigned char = long	Preserve low-order byte
unsigned short = long	Preserve low-order bytes
unsigned long = long	Preserve pattern; high-order bit loses function as sign bit

SHIFT AND CAST EXAMPLE

- I found this example in the book.
- funExample.c

OVERFLOW EXAMPLES

- It is also important that you know what you are doing when, adding, subtracting, incrementing and decrementing signed and unsigned numbers.
- As we wrap up unsigned/signed and expansion, let's talk a little about overflow.
- In computer programming, an integer overflow occurs when an arithmetic operation attempts to create a numeric value that is outside the range that can be represented with a given number of digits - either higher than the maximum or lower than the minimum representable value.
- CODE: overFlow2.c, overflowEx.c, overFlow.c

2.3 UNSIGNED ADDITION

- When x and y are unsigned int and $x + y > 2^w - 1$ (the unsigned int max of w), the sum overflows.
- Ex: (w = 4) "w" in this case, is the word size or number of bits. Consider x (an unsigned 4 bit number), x = 9 + 12 (both are 4 bits). Adding these together will give us a 5 bit number.
- Truncation rule says x would result in a mod 16 (range of a 4 bit number is 0 - 15)
  - 9+12=21 which is outside the range of 0-15, so 21%16 = 5
  - This is the same as 1001 + 1100 = 10101%16 = 0101 (same as 10101 truncating any bits above 4 bits)

2.3 UNSIGNED ADDITION - DETECTING AN OVERFLOW FOR UNSIGNED ADDITION

- **Detecting an overflow for unsigned addition**
  - **If we have a number with 4 bits (w = 4), the maximum value of this number is $2^4 - 1 = 15$ (1111). Visually we can see that 9 + 12 = 21 which is larger than 15 so this is an indication of an overflow.**
  - **Determining overflow with code. From the previous slide, if there is an overflow what is going to happen? The higher order bits are going to be truncated. Ex. 1001 = 9 added to 1100 = 12 will give me 10101%16 will give me 0101 = 5.**
  - **Principal: Detecting overflow of unsigned addition**
    - **For x and y in the range 0 <= x, y <= Umax of w, let s = x + y. Then we can say 's' overflowed if and only if s < x (or equivalently, s < y).**
    - **As illustrated in the previous example, we saw that (unsigned 4 bit number) 9 + 12 = 5, we can see that overflow occurred since 5 < 9 as well as 5 < 12. In layman's terms if the result is less than either of the values being added then there is a problem.**

2.3.2 TWO'S COMPLEMENT ADDITION

- **With 2's complement addition you can have two types of overflow**
  - **Negative overflow**
  - **Positive overflow**
- **We will discuss the definition of each then look at an example.**

UMin $= 0$
UMax $= 2^w - 1$

TMin $= -2^{w-1}$
TMax $= 2^{w-1} - 1$

OVERFLOW

TMin $= -2^{w-1}$
TMax $= 2^{w-1} - 1$

- **Negative:**
  - **When $x + y < -2^{w-1}$ (TMin), there is a negative overflow.**
  - **This gives you 2^w more than the integer sum**
- **Detecting an overflow in 2's complement addition:**
  - **For x and y in the range $TMIN_w <= x,y <= TMAX_w$ let s = x+y of size w. The computation has had a negative overflow if and only if x < 0 and y < 0, but s >= 0.**

Examples of negative overflow
w = 4, 2^4 = 16, range (-8 to 7)

x	y	x + y	x + y trunc to 2^4
-8	-5	-13	3
1000	1011	10011	0011
-8	-8	-16	0
1000	1000	10000	0

Negative Overflow
$= (x+y) + 2^w$

OVERFLOW

- **Positive:**
  - **When $x + y >= 2^{w-1}$, there is a positive overflow.**
  - **This gives you 2^w less than the integer sum.**
- **Detecting overflow in 2's complement addition:**
  - **For x and y in the range $TMIN_w <= x, y <= TMAX_w$ let $s = x + y$ of size w. Then the computation of s has had a positive overflow if and only if $x > 0$ and $y > 0$ but $s <= 0$**

$TMin = -2^{w-1}$

$TMax = 2^{w-1} - 1$

Example of positive overflow

$w = 4$, $2^4 = 16$, range (-8 to 7)

x	y	x + y	x + y trunc to 2^4
5	5	10	-6
0101	0101	01010	1010

Positive overflow $= (x+y) - 2^w$

OVERFLOW

No overflow
w = 4

x	y	x + y	x + y trunc to 2^4
-8	5	-3	-3
1000	0101	11101	1101
2	5	7	7
0010	0101	00111	0111

W = 5

PRACTICE

X	Y	X+Y	Truncated X + Y	Case
-12	-15	-27	5	Negative Overflow
10100	10001	100101	00101	
-8	-8	-16	-16	No Overflow
11000	11000	110000	10000	
-9	8	-1	-1	No Overflow
10111	01000	111111	11111	
2	5	7	7	No Overflow
00010	00101	000111	00111	
12	4	16	-16	Positive
01100	00100	010000	10000	

UNSIGNED MULTIPLICATION AND THE IMPLICATIONS IN C

- **Range is $0 <= x, y <= 2^w - 1$ the product of x and y has a range of 0 to $(2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$**

- **This requires 2w bits, so for 8 bits this would be a 16 bit number. However the higher bits would be truncated to 2^w.**

  - **Ex. If $w = 4 = (2^8 - 2^5 + 1) = 256 - 32 + 1 = 225$, since $1111 = 15_{10}$, then 15 * 15 = 225 or 11100001 this will be truncated to 2^4 = 225 mod 16 or (11100001)mod 16 = 0001**

  - **For x and y such that $0 <= x,y <= Umax_w = (x*y) \bmod 2^w$**

```
                1111
           x    1111
                1111
               11110
              111100
             1111000
=           11100001   = 225 mod 16 = ????
```

Bit-level representation of the product operation is identical for both unsigned and two's complement multiplication.

Let's look at a chart.

MULTIPLICATION

- **Ex. w = 3 therefore the x * y is 6 bits the overflow is truncated to 3**
- **This chart shows that the bit-level representation of the product operation is identical for both unsigned and two's-complement multiplication (after truncation by 2^3)**

The bit representation is the same for unsigned and signed (2's Complement)

Modes	X		Y		X*Y		Truncated	X*Y
unsigned	5	101	3	011	15	001111	7	111
2's comp	-3	101	3	011	-9	110111	-1	111
unsigned	4	100	7	111	28	011100	4	100
2's comp	-4	100	-1	111	4	000100	-4	100
unsigned	3	011	3	011	9	001001	1	001
2's comp	3	011	3	011	9	001001	1	001

CODE: multiplyTest.c (you can look at this)

MORE ON MULTIPLICATION

- Historically, integer multiplication instructions on many machines were fairly slow > 10 clock cycles to complete. However, other integer operations — addition, subtraction, bit-level operations, shifts — require ~1 clock cycle. More recently intel Core i7 takes 3 or more clock cycles for integer multiplication.

- Therefore, compilers are likely to optimize by attempting to replace multiplications by constant factors with combinations of shift and addition. However, the combination of shifts and additions or subtractions could become quite expensive as well, therefore the compiler will make the decision if this technique is worth the optimization.

MULTIPLICATION BY POWERS OF 2

- When covering left and right shift we saw that we can use left shift to multiply by a power of 2.
- As discussed, multiplication is more costly than shifting, addition, and subtraction

Ex. x * 14 is the same as $2^3 + 2^2 + 2^1 = 14$ (8 + 4 + 2 = 14) so we can say (x << 3) + (x << 2) + (x << 1) = x * 14

Lets say x = 3;

3 * 14 = 42
00000011 << 3 = 00011000
00000011 << 2 = 00001100
00000011 << 1 = <u>00000110</u>
= 00101010 = 42

This can also be done using subtraction.

MULTIPLICATION BY SHIFT ADDITION/SUBTRACTION

- The example we saw in the previous slide multiplied x * 14 using 3 shifts and 2 additions. We will now see that we can do the same calculation using 2 shifts and 1 subtraction.

- Suppose x = 3 and x * 14, since $14 = 2^4 - 2^1 = 16 - 2$, we can do the following:

(x << 4) - (x << 1) = x*14 = 3 * 14 = 42

00000011 << 4 = 00110000

00000011 << 1 = 00000110

```
   00110000
-  00000110
   00101010 = 42
```

PRACTICE

For each of the following values of K, find ways to express x * K using the specified number of operations.
CODE: shiftTest.c

K	# of Shifts	# of Adds/Subs	Expression
6	2	1	
31	1	1	
55	2	2	

DIVISION

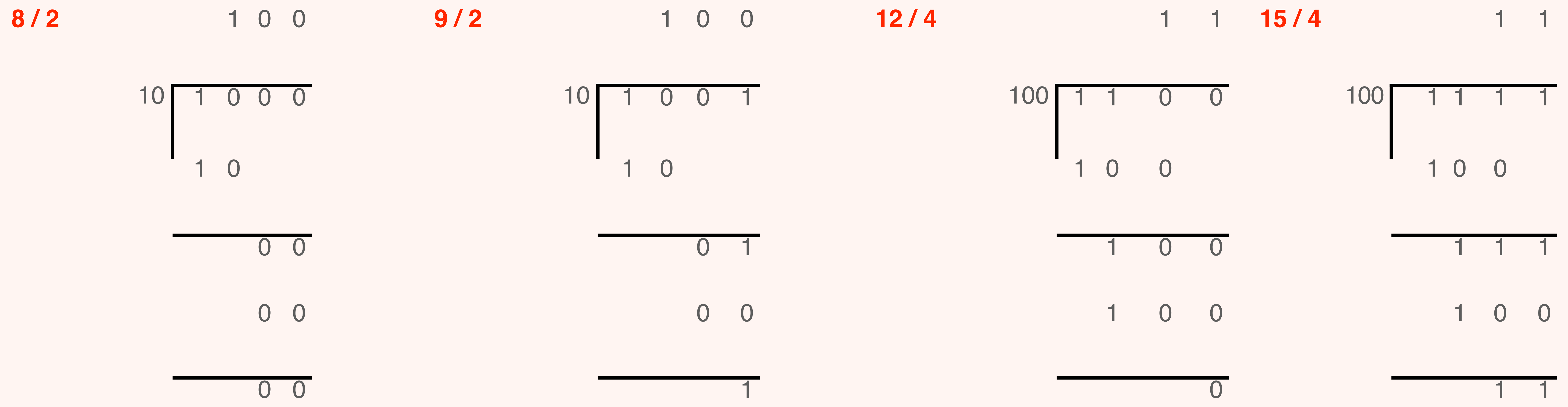
ROUNDING REVIEW

- The chart to the right is a quick review of rounding and what it means to round down/up and Rounding towards zero/away from zero.

y	round down (towards −∞)	round up (towards +∞)	round towards zero	round away from zero	round to nearest
+23.67	+23	+24	+23	+24	+24
+23.50	+23	+24	+23	+24	+24
+23.35	+23	+24	+23	+24	+23
+23.00	+23	+23	+23	+23	+23
0	0	0	0	0	0
−23.00	−23	−23	−23	−23	−23
−23.35	−24	−23	−23	−24	−23
−23.50	−24	−23	−23	−24	−24
−23.67	−24	−23	−23	−24	−24

DIVIDING BY POWER OF 2

- **Division on most machines are even slower than integer multiplication 30+ machine cycles**
- **Division by a power of 2 can be performed by using right shift**
  - **Signed/unsigned will determine if the shift is logical or arithmetic**



DIVIDING BY POWER OF 2 - UNSIGNED

- **For C variables x and k with unsigned values x and k, such that 0<=k<=w, the C expression x >> k yields the value = $x/2^k$**
- **Unsigned performing a logical shift by k has the same effect as dividing by 2^k and then rounding TOWARD ZERO**

K	X >> K (binary)	Decimal	$12{,}340/2^k$
0	0011000000110100	12340	12340.0
1	0001100000011010	6170	6170.0
4	0000001100000011	771	771.25
8	0000000000110000	48	48.203125

DIVIDING BY POWER OF 2 TWO'S COMPLEMENT

- **Two's complement division by a power of 2, rounds down**
- **For C variables x and k with 2's complement value for x and unsigned value of k, such that 0 <= k < w, the C expression x >> k when the right shift performed is arithmetic yields the value = $x/2^k$**
- **If the most significant bit is 0 (non-negative) the procedure is the same as a logical shift**
- **The effect of an arithmetic right shift on a negative number has the same effect of division by power of 2 except it ROUNDS DOWN rather than TOWARD ZERO**

K	X >> K (binary)	Decimal	$-12,340/2^k$
0	1100111111001100	-12340	-12340.0
1	**1**110011111100110	-6170	-6170.0
4	**1111**110011111100	-772	-771.25
8	**11111111**11001111	-49	-48.203125

DIVIDING BY POWER OF 2 TWO'S COMPLEMENT

- Rounding errors can accumulate. We can correct for the improper round that occurs when a negative number is shifted right, by "biasing" the value before shifting.

- For C variables x and k with 2's complement value x and unsigned value of k, such that 0<= k < w, the C expression (x + (1<<k) -1) >> k, when the arithmetic right shift is performed it yields the value = $x/2^k$

- By adding a bias before the right shift, the result is rounded TOWARD ZERO

K	Bias	-12340 + bias (binary)	>> K (binary)	Decimal	$-12{,}340/2^k$
0	0	1100111111001100	1100111111001100	-12340	-12340.0
1	1	1100111111001101	1110011111100110	-6170	-6170.0
4	15	1100111111011011	1111110011111101	-771	-771.25
8	255	1101000011001011	1111111111010000	-48	-48.203125

- (X < 0 ? X + (1<<k) - 1 : X) >>k

  Add bias

SUMMARY

- Through out this chapter we have seen how the computer handles various “integer” arithmetic
- The finite word size used by computers limited the range of possible values, therefore overflows can happen.
- We have seen 2’s complement representation provides a clever way to represent both negative and positive numbers, while using the same bit-level implementations as are used to perform unsigned arithmetic (+,-, *, /)
- We have also seen how each of these operations can produce unexpected results and not so easy to find bugs in C code.
- As a matter of fact some of these bugs can stump even the most experienced C Programmers
- It is important that we have at least some understanding of how our data is stored in the computer.

FLOATING POINT

FRACTIONAL DECIMAL NUMBERS

- We know that fractional decimal numbers are represented by
  - $d_m\, d_{m-1} \ldots d_1 d_0 \,.\, d_{-1} d_{-2} \ldots d_{-n}$ (12.34) where each digit d_i has a range of 0 – 9, the weight of the number is relative to the decimal point. The numbers to the left are non-negative powers of 10 and the numbers on the right are weighted negative powers of 10 representing the fractional part of the number.
  - For Example: $12.34 = (1 * 10^1 + 2 * 10^0) + (3 * 10^{-1} + 4 * 10^{-2}) = 12\ 34/100$ (12.34)

Whole part

Fractional part

FRACTIONAL BINARY NUMBERS
A.K.A. POSITIONAL NOTATION

- **Fractional binary are represented the same way**
  - **$b_m\, b_{m-1} \ldots\, b_1\, b_0\, .\, b_{-1} b_{-2} \ldots d_{-n+1} d_{-n}$ The "." is known as the binary point rather than decimal point with the bits to the left being weighted by nonnegative powers of 2, and those on the right being weighted by negative powers of 2.**

For example 101.11_2 represents the number

$$\underbrace{(1 * 2^2 + 0 * 2^1 + 1 * 2^0)}_{\text{Whole part}} + \underbrace{(1 * 2^{-1} + 1 * 2^{-2})}_{\text{Fractional part}} = 4 + 0 + 1 + 1/2 + 1/4 = 5\ 3/4 \quad (101.11)$$

FRACTIONAL BINARY NUMBERS A.K.A. POSITIONAL NOTATION

- **Decimal (base 10 notation can not exactly represent numbers such as 1/3 = .3333... and 5/7 = .7142871428 and so on**
- **Similarly, fraction binary notation can only represent numbers that can be written in the following format - $x * 2^Y$ other values can only be approximated**

FRACTIONAL BINARY NUMBERS

Example:

$1/5_{10}$ - can be represented exactly as a fractional decimal number .20

As a fractional binary we can not represent it exactly, instead we approximate it with increasing accuracy by lengthening the binary representation:

Binary representation	Value	Decimal
0.0	0/2	0.0
0.01	1/4	0.25
0.010	2/8	0.25
0.0011	3/16	0.1875
0.00110	6/32	0.1875
0.001101	13/64	0.203125
0.0011010	26/128	0.203125
0.00110011	51/256	0.19921875

IEEE FLOATING POINT

HISTORICAL PERSPECTIVE OF FRACTIONAL BINARY NUMBERS

- Until the late 1970's - early 1980's each manufacturer handled floating point numbers differently
- ~1976 Intel started designing the 8087 - a chip that provided support for the 8086 processor
- Intel hired Dr. William Kahan, a professor from Berkley, as a consultant to help develop the floating-point standard for Intel's future chips.
- Kahan joined forces with an IEEE committee that later adopted the floating-point standard Kahan was developing for intel.
- This is what we now know as IEEE standard 754

IEEE FLOATING-POINT REPRESENTATION

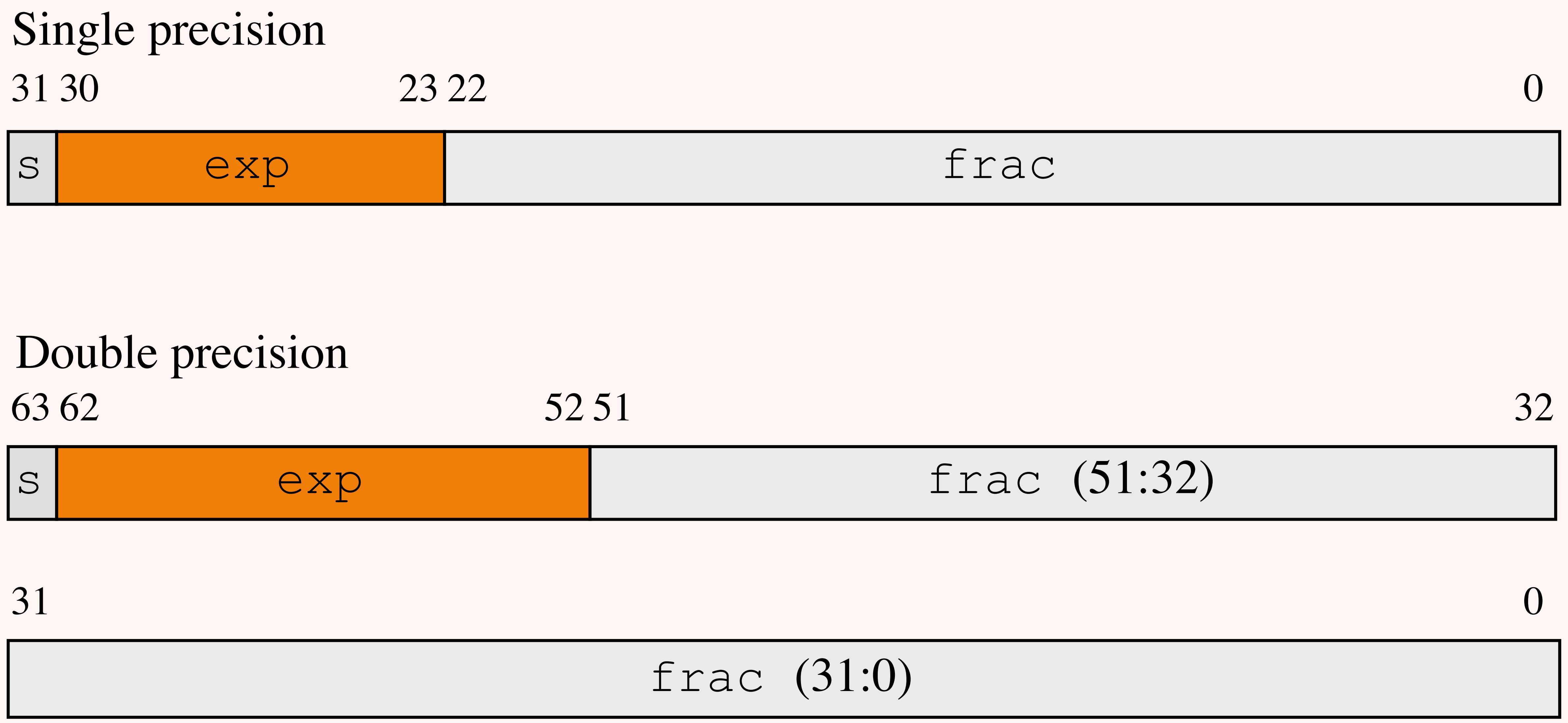
- Positional notation such as considered in the previous section would not be efficient for representing very large numbers. $5 * 2^{100}$ would be 101 followed by 100 (0)'s.
- The IEEE floating point standard allows us to represent numbers in the following form
  - S = the sign bit
  - E = the exponent
  - M = the significand (mantissa)

IEEE FLOATING-POINT REPRESENTATION

Number of bits for each section:

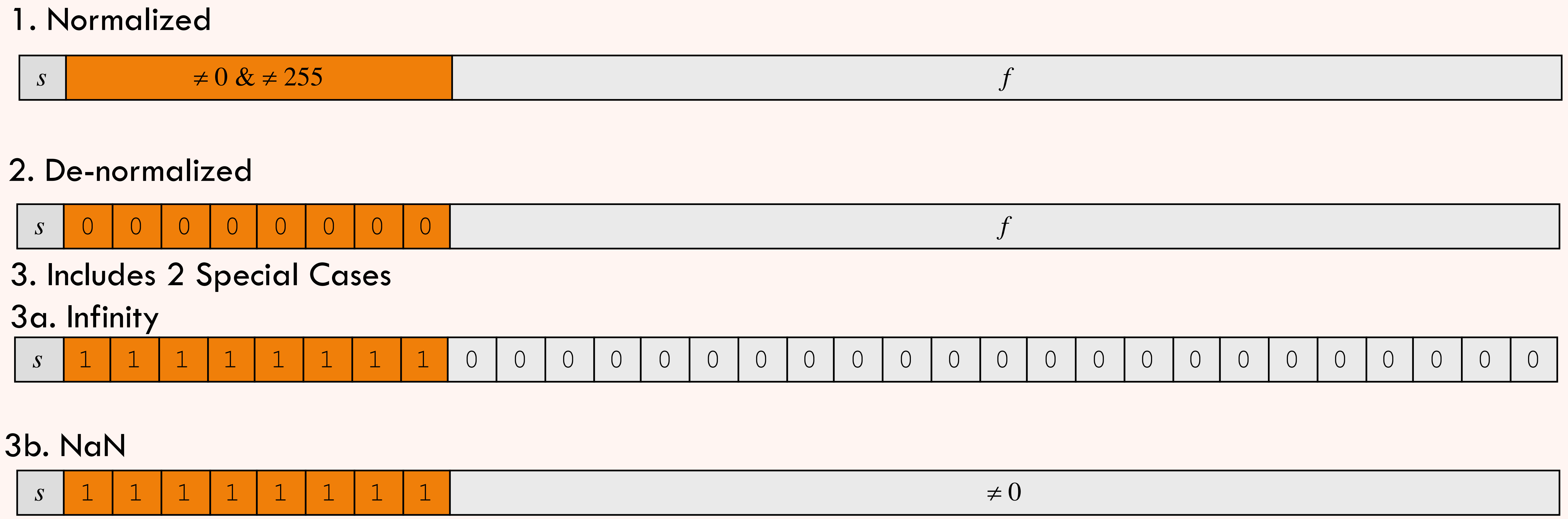
Single precision:
s=1 (sign) k=8 (exp) n=23 (frac)
Bias of 127

Double precision
s=1 (sign) k=11 (exp) n=52 (frac)
Bias of 1023



IEEE FLOATING-POINT REPRESENTATION

- **The value encoded by a given bit representation can be divided into three different cases (the latter having two variants), depending on the value of frac.**



IEEE FLOATING-POINT REPRESENTATION NORMALIZED

1. Normalized

s	$\neq 0$ & $\neq 255$	f

- **This is the most common case.**
- **Is used when the bit pattern of exponent portion is neither all 0's nor 1's**
- **The exponent E = e - Bias, where e is the unsigned number with bit number $e_{k-1}...e_1\ e_0$ and Bias is a value equal to $2^{k-1} – 1$, which is 127 for single precision and 1023 double precision. This yields exponent ranges from -126 thru 127 for single precision and -1022 thru 1023 for double precision**
- **The frac represents the fractional part of the number**
  - **The significand (mantissa)**

IEEE FLOATING-POINT REPRESENTATION DENORMALIZED

2. De-normalized

s	0	0	0	0	0	0	0	0	f

- **This case is used when the exponent field is 0**
- **Serves 2 purposes**
  - **Provides a way to represent number value 0, +0.0 has a bit pattern of all zeros including S. If the sign bit is 1 and M = *f* = 0 then we have -0.0. In IEEE FP standard -0.0 and +0.0 are sometimes considered different??**
  - **Provides a way to represent numbers vary close to 0.0 they provide a property known as gradual underflow in which possible numeric values are spaced evenly near 0.0**

IEEE FLOATING-POINT REPRESENTATION SPECIAL CASE

3a. Infinity

s	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

3b. NaN

s	1	1	1	1	1	1	1	1	$\neq 0$

- **These occur when the E = all 1's**
  - **If the frac fields are all 0's this represents positive infinity when S = 0 or negative infinity with S = 1. Infinity can represent results that overflow, as when we multiply two very large numbers or divide by 0**
  - **When the frac is not all 0's then this results in the dreaded NaN - sometimes used when we have uninitialized data.**
  - **https://evanw.github.io/float-toy/**

IEEE FLOATING POINT

- Although this floating point scheme is more complex then positional notion described in previous slides, the IEEE floating point standard provides additional flexibility for representing a wide range of values. Despite the flexibility, a floating point format with a constant set of bits can not precisely represent all possible values. Therefore there could still be rounding errors.
- Couple examples I found:
  - During the Gulf War of 1991, a rounding error caused an American Patriot missile to fail to intercept an Iraqi missile, resulting in killing 28 soldiers and injuring many more.
  - In 1996, the European Space Agency's first launch of a rocket exploded 39 seconds after take off. The rocket used a great deal of code from a previous rocket triggering an overflow when attempting to convert a floating-point value into an integer value.