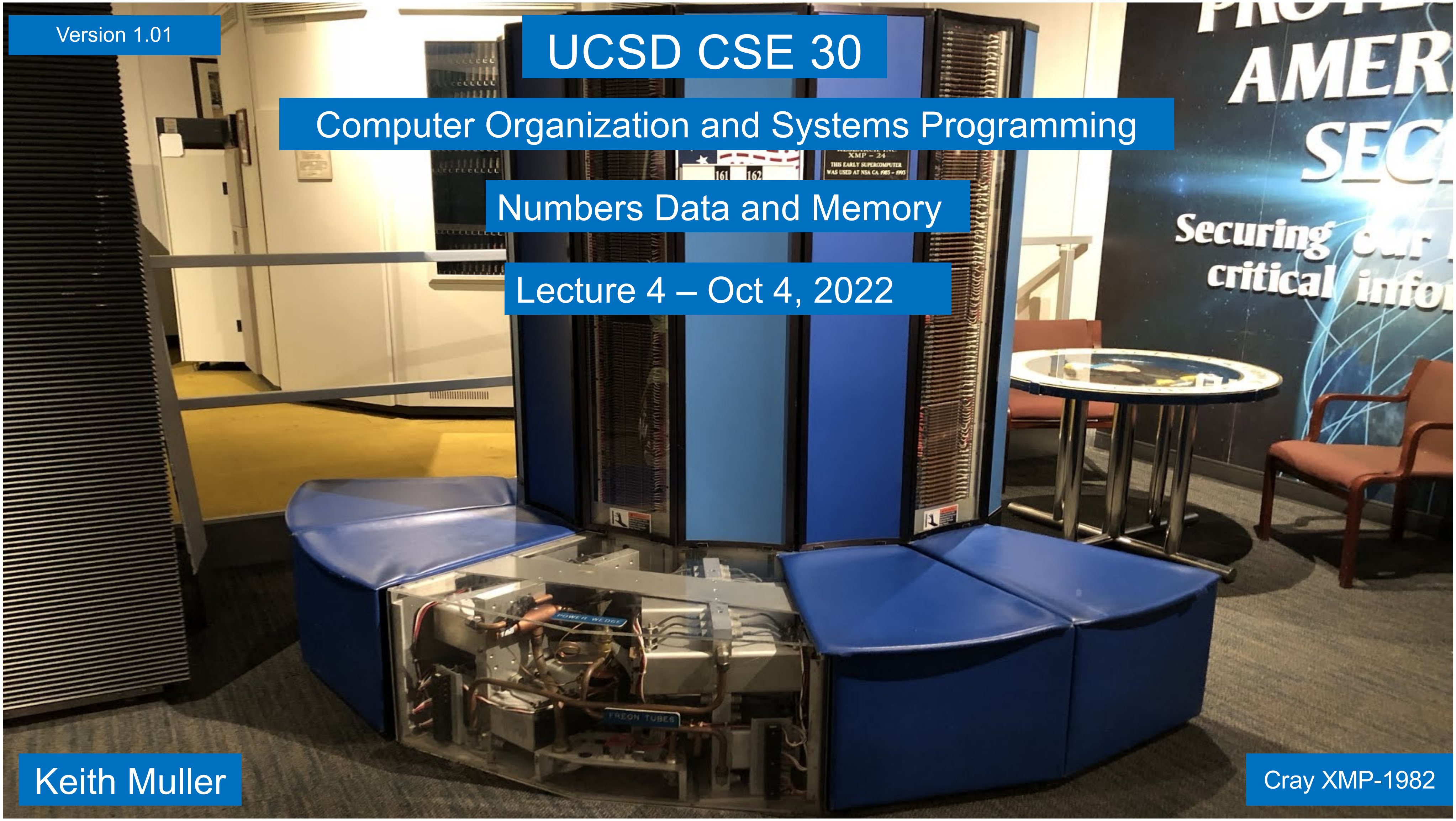
Version 1.01

UCSD CSE 30

Computer Organization and Systems Programming

Numbers Data and Memory

Lecture 4 – Oct 4, 2022



Keith Muller

Cray XMP-1982

Review: Decimal Numbering

- Decimal is base 10
  - from "decem" (Latin) ⇒ Ten Characteristics
- Ten symbols (Why?)

  0 1 2 3 4 5 6 7 8 9
- **How do we represent larger numbers?**
  - **Large numbers are a sequence of digits**
  - Each digit is one of the available symbols
  - n-digit numbers as coefficients in a polynomial

Digit position

$$d_{n-1}\ d_{n-2} \ldots d_1\ d_0 = d_{n-1} \bullet 10^{n-1} + d_{n-2} \bullet 10^{n-2} + \ldots + d_1 \bullet 10 + d_0$$

symbol value

- Examples
  - 7061 in decimal (base 10)
  - $7061_{10} = (7 \times 10^3) + (0 \times 10^2) + (6 \times 10^1) + (1 \times 10^0)$

x

Review: Binary Numbering

- Binary is base 2
  - *adjective:* being in a state of one of two **mutually exclusive** conditions such as **on** or off, true or false, molten or frozen, presence or absence of a signal
  - From Late Latin *bīnārius* (“consisting of two”)
- Two symbols:

  0 1

- Numbers in C starting with 0b are binary
- Example: What is 0b110 in base 10?
  - 0b110 = $110_2 = (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) = 6_{10}$

powers of two

- A **bit** is a single binary digit
- A **byte** is an 8-bit value

$$Unsigned\ binary\ Number = \sum_{i=0}^{i=n-1} b_i\ x\ 2^i = b_{n-1}2^{N-1} + b_{n-2}2^{N-2} + \ldots + b_1 2^1 + b_0 2^0$$

x

Review: Octal Numbering

- Eight symbols

  0, 1, 2, 3, 4, 5, 6, 7

  Notice that we no longer use 8 or 9

- Base comparison:
  - Base 10: `0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12…`
  - Base 8: `0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14…`

- Numbers in C starting with a 0: 07061, are octal

- Example: What is 07061_8 in base 10?
  - $07061_8 = (7 \times 8^3) + (0 \times 8^2) + (6 \times 8^1) + (1 \times 8^0) = 3633_{10}$

powers of eight

subscript indicates base

in C leading 0 indicates octal

$$Unsigned\ octal\ Number = \sum_{i=0}^{i=n-1} b_i\ x\ 8^i = b_{n-1}8^{N-1} + b_{n-2}8^{N-2} + \ldots + b_1 8^1 + b_0 8^0$$

x

Review: Hexadecimal Numbering

- hexadecimal is base 16
  - From "hexa" (Ancient Greek ἑξα-) ⇒ six
  - and from "decem" (Latin) ⇒ ten
- Sixteen symbols

  0 1 2 3 4 5 6 7 8 9 a b c d e f

- Numbers in C starting with 0x are hexadecimal
  - 16_{10} = 0x10_{16}
- Example: What is 0xa5 in base 10?
  - 0xa5 = $a5_{16}$ = $(10 \times 16^1) + (5 \times 16^0) = 165_{10}$
- Hexadecimal numbers are very commonly used in programming to express binary values
  - Imagine the difficulty in correctly expressing a 64-bit binary value in your code

$$Unsigned\ Hex\ Number = \sum_{i=0}^{i=n-1} b_i\ x\ 16^i = b_{n-1}16^{N-1} + b_{n-2}16^{N-2} + \ldots + b_1 16^1 + b_0 16^0$$

x

Number Base Overview (as written in C)

- Decimal is base 10, Hexadecimal is base 16, and octal is base 8
- **Octal digits** have 8 values 0 – 7 (written in C as **0**0 – **0**7, careful **0**73 is octal = 59 in decimal)
- **Hex digits** have 16 values 0 - 9 a - f (written in C as 0x0 – 0xf)
- No standard prefix in C for binary (most use hex) – gcc (compiler) allows 0b prefix others might not

Hex digit Octal digit	0x0 00	0x1 01	0x2 02	0x3 03	0x4 04	0x5 05	0x6 06	0x7 07
Decimal value	0	1	2	3	4	5	6	7
Binary value	0b0000	0b0001	0b0010	0b0011	0b0100	0b0101	0b0110	0b0111

Hex digit Octal digit	0x8 010	0x9 011	0xa 012	0xb 013	0xc 014	0xd 015	0xe 016	0xf 017
Decimal value	8	9	10	11	12	13	14	15
Binary value	0b1000	0b1001	0b1010	0b1011	0b1100	0b1101	0b1110	0b1111

x

Binary <---> Hexadecimal Equivalences

- Hex → Binary: $16^1 = 2^4$ 1 digit hex = 4 digits binary
  1. Replace hex digits with binary digits
  2. drop leading zeros
  - Example: 0x2d to binary
    - 0x2 is 0b00**10**, 0xd is 0b1101
    - Drop two leading zeros, answer is 0b**10**1101
- Binary → Hex: $2^4 = 16^1$
  1. Pad with enough leading zeros until number of digits is a multiple of 4
  2. replace each **group of 4** with the **HEX equivalent**
  - Example: 0b101101
    - **Pad on the left** to: 0b 0010 1101
    - Replace to get: 0x2d

Base 10	Base 2	Base 16
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	a
11	1011	b
12	1100	c
13	1101	d
14	1110	e
15	1111	f

x

Hex to Binary (group 4 bits per digit from the right)

- Each Hex digit is 4 bits in base 2 $16^1 = 2^4$

0x f a 5 3

1111 1010 0101 0011

0b1111101001010011

binary start with a 0b in C

x

Binary to Hex (group 4 bits per digit from the right)

- 4 binary bits is one Hex digit $2^4 = 16^1$

$$0b\ \underbrace{0110}_{6}\ \underbrace{1010}_{a}\ \underbrace{0011}_{3}\ \underbrace{1111}_{f}$$

$$\boxed{0x6a3f}$$

hex start with 0x in C

Binary to Octal (group 3 bits per digit from the right)

- 3 binary bits is one Octal digit $2^3 = 8^1$

$$0b\ \underbrace{0}_{0}\ \underbrace{110}_{6}\ \underbrace{101}_{5}\ \underbrace{000}_{0}\ \underbrace{111}_{7}\ \underbrace{111}_{7}$$

$$\boxed{065077}$$

octal start with a 0 in C

x

Octal to Binary (group 3 bits per digit from the right)

- One Octal digit is three binary digits $2^3 = 8^1$

$$\underbrace{1}_{01}\ \underbrace{111}_{7}\ \underbrace{101}_{5}\ \underbrace{001}_{1}\ \underbrace{010}_{2}\ \underbrace{011}_{3}$$

$\boxed{\text{0b1111101001010011}}$

binary start with a 0b in C

x

Looking at the Powers of Two

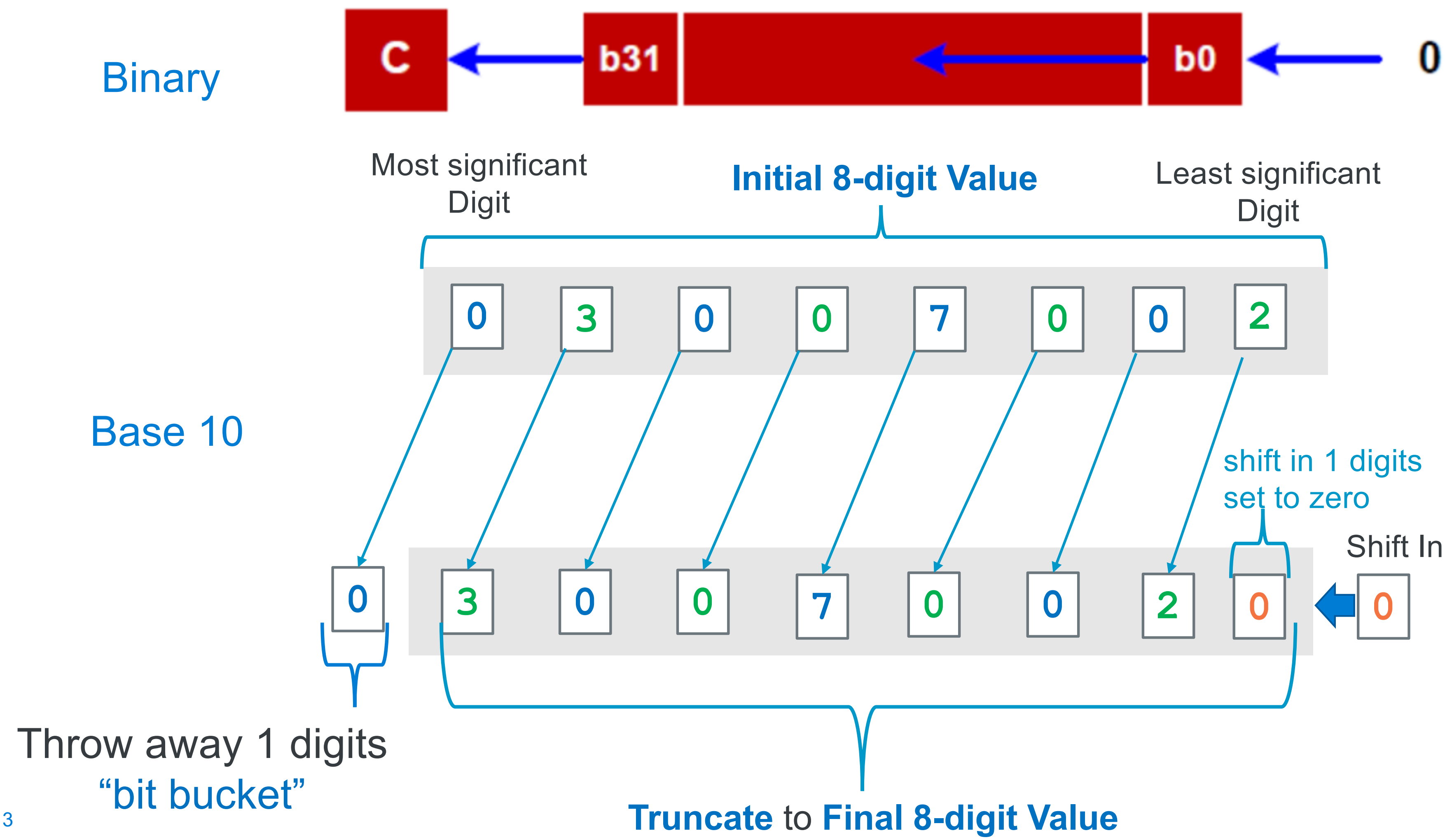
$$Unsigned\ binary\ Number = \ b_{n-1}2^{N-1} + \ b_{n-2}2^{N-2} + \ \ldots + \ b_1 2^1 + \ b_0 2^0$$

Base 10	$b_5 \cdot 32$	$b_4 \cdot 16$	$b_3 \cdot 8$	$b_2 \cdot 4$	$b_1 \cdot 2$	b_0
5	0	0	0	1	0	1
10	0	0	1	0	1	0
20	0	1	0	1	0	0
40	1	0	1	0	0	0

- Multiple – left shift ←
- Divide – right shift →

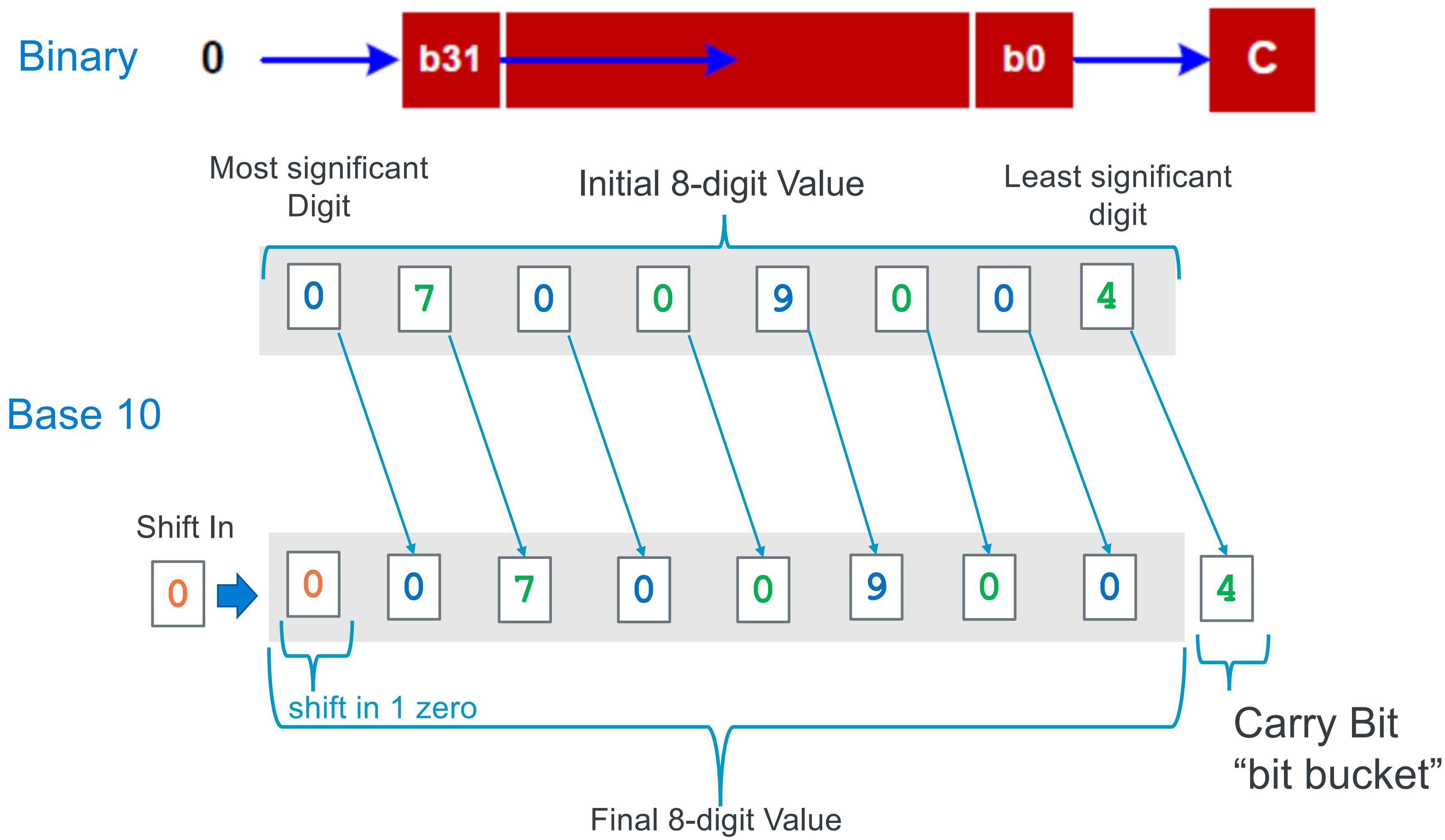
x

Multiply By Base: Shift Left 1 digits = Multiply by 2



x

Divide by Base: Shift Right 1 Digit = Divide by 2



x

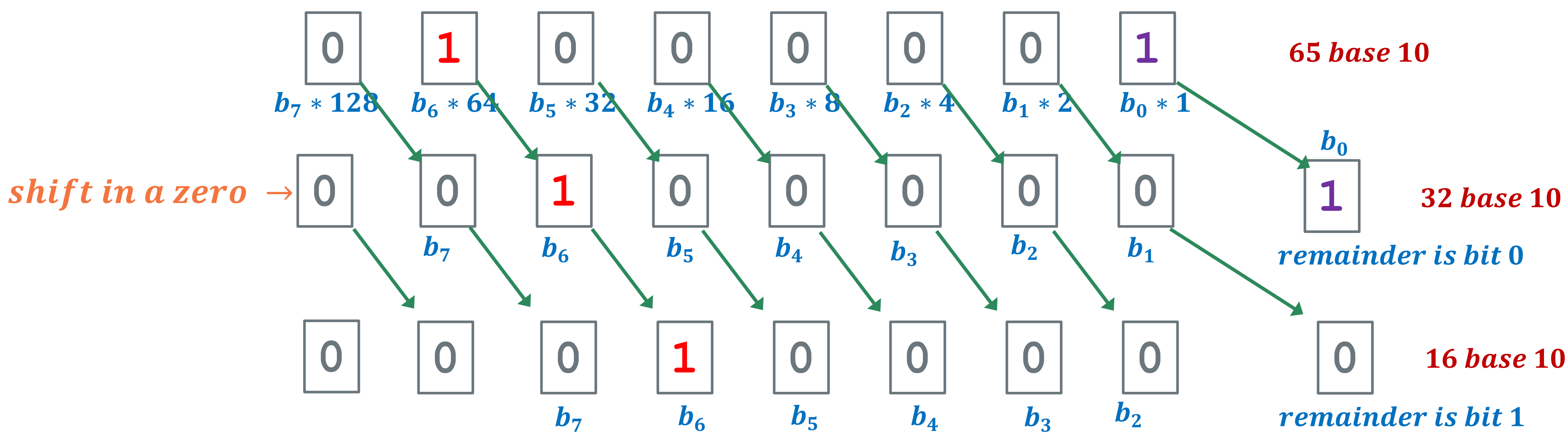
Use a Right Bit Shift: Unsigned Decimal To Unsigned Binary

Here is an Algorithmic approach to convert unsigned numbers to binary

Perform a sequence of **divisions** (**right shift**) to **isolate the remainder**

$$\boxed{Unsigned\ binary\ Number\ = \quad b_{n-1}2^{N-1} + \ b_{n-2}2^{N-2} + \cdots + \ b_1 2^1 + \ b_0 2^0}$$

$$\boxed{Unsigned\ Binary\ Number = \ 2\ x\ (\cdots\ (2\ x\ (2\ x\ b_{n-1} \ + \ b_{n-2}))\ + \cdots + \ b_1\ ) + \ \boldsymbol{b_0}}$$



x

Unsigned Decimal to Unsigned Binary Conversion

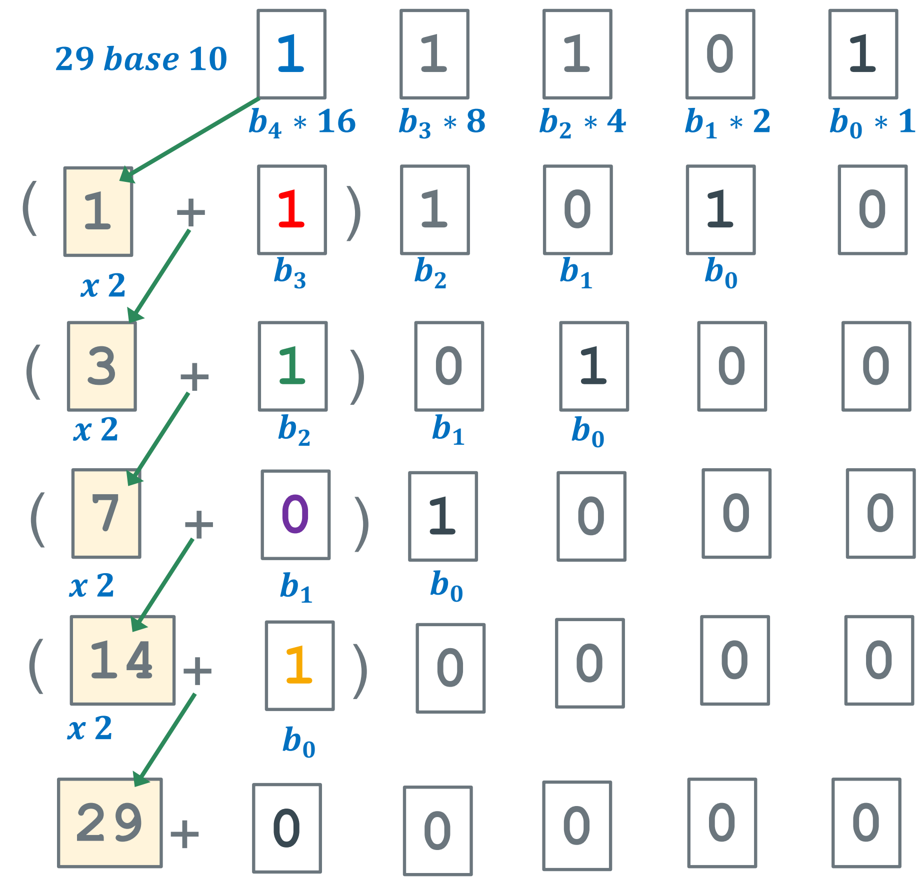
dividend 249	Quotient	Remainder	Bit Position
249/2	124	1	b0
124/2	62	0	b1
62/2	31	0	b2
31/2	15	1	b3
15/2	7	1	b4
7/2	3	1	b5
3/2	1	1	b6
1/2	0	1	b7

249(base 10) = $b_7\, b_6\, b_5\, b_4\, b_3\, b_2\, b_1 b_0 = 0b11111001$

11111001 = (1x128) + (1x64) + (1x32) + (1x16) + (1x8) + 1 = 249

x

Left Bit Shift & add: Unsigned Binary to Unsigned Decimal



$$b_{n-1}2^{N-1} + \ b_{n-2}2^{N-2} + \cdots + \ b_1 2^1 + \ b_0 2^0$$

- Base conversion via a sequence of n multiplications (left shift) and n additions
  - 111 base 2 -> ((1x2 + 1) x 2) + 1
- Alternatively, you can memorize and use the positional weights to convert

- 11101 base 2 = (1 x 16) + (1 x 8) + (1 x 4) + (0 x 2) + (1 x 1) = 29

x

Unsigned Binary to Unsigned Decimal Conversion

What is $\overset{b_7}{0}\ \overset{b_6}{1}\ \overset{b_5}{1}\ \overset{b_4}{0}\ \overset{b_3}{0}\ \overset{b_2}{1}\ \overset{b_1}{0}\ \overset{b_0}{1}$ (base 2) in decimal (N)?

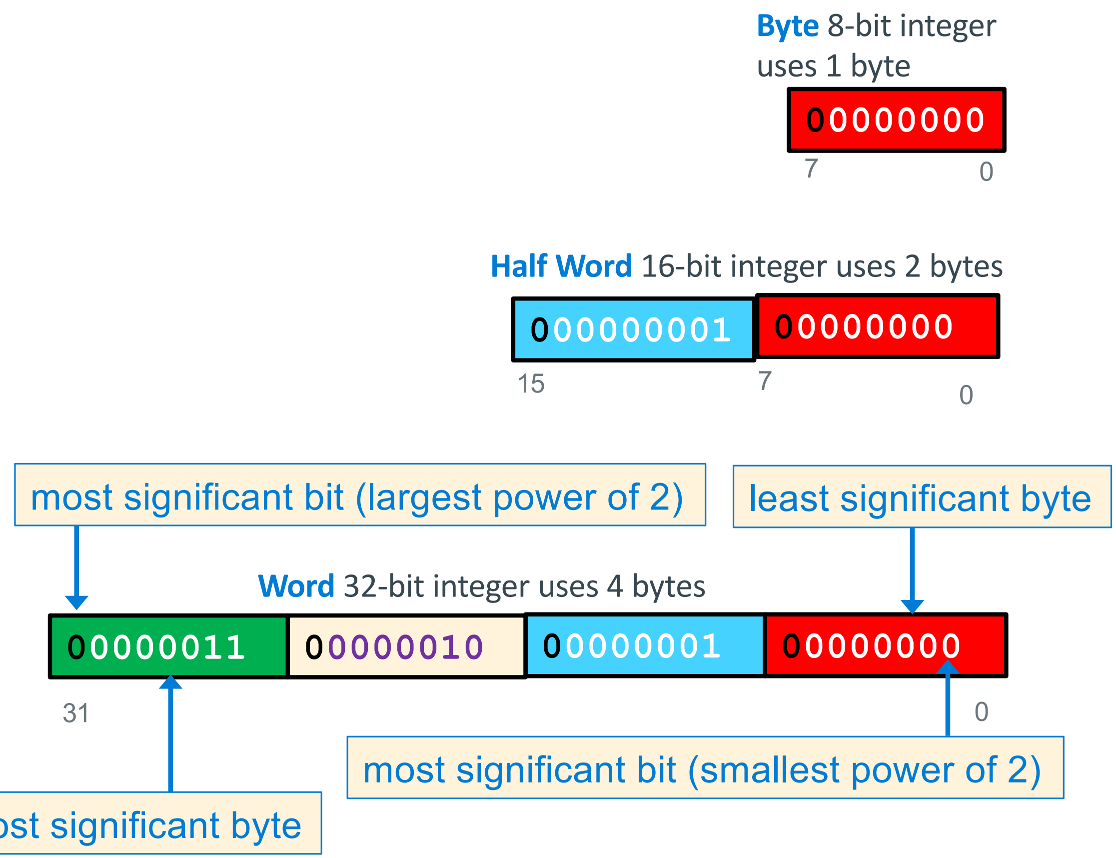
Product Shift Left	Addend	Bit Position	Product
0	+ 0	b7	0
2 x 0 = 0 (shift left)	+ 1	b6	1
2 x 1 = 2	+ 1	b5	3
2 x 3 = 6	+ 0	b4	6
2 x 6 = 12	+ 0	b3	12
2 x 12 = 24	+ 1	b2	25
2 x 25 = 50	+ 0	b1	50
2 x 50 = 100	+ 1	b0	101

101 (base 10) = (1x64) + (1x32) + (1x4) + 1 (checking the conversion)

x

Different Type of Numbers each have a Fixed # of Bits
Spanning one or more contiguous bytes of memory

C Data Type	AArch-32 contiguous Bytes
char (arm unsigned)	1
short int	2
unsigned short int	2
int	4
unsigned int	4
long int	4
long long int	8
float	4
double	8
long double	8
pointer *	4



x

Variables: Size

- **Integer types**
  - **char**, **int**
- **Floating Point**
  - **float**, **double**
- Modifiers for each base type
  - **short** [int]
  - **long** [int, double]
  - **signed** [char, int]
  - **unsigned** [char, int]
  - **const**: variable read only

C Data Type	AArch-32 contiguous Bytes	AArch-64 contiguous Bytes	printf specification
char (arm unsigned)	1	1	%c
short int	2	2	%hd
unsigned short int	2	2	%hu
int	4	4	%d / %i
unsigned int	4	4	%u
long int	4	8	%ld
long long int	8	8	%lld
float	4	4	%f
double	8	8	%lf
long double	8	16	%Lf
pointer *	4	8	%p

size of a pointer is the word size

- char type
  - One byte in a byte addressable memory
  - **Signed** vs **Unsigned** Char implementations
  - **Be careful** char is unsigned on arm and signed on other HW like intel

x

Fixed size types in C

- Sometimes programs need to be written for a particular range of integers or for a particular size of storage, regardless of what machine the program runs on
- We will need to do this in PA3
- In the file **`<stdint.h>`** the following fixed size types are defined for use in these situations:

Signed Data types	Unsigned Data types	Exact Size
`int8_t`	`uint8_t`	8 bits (1 byte)
`int16_t`	`uint16_t`	16 bits (2 bytes)
`int32_t`	`uint32_t`	32 bits (4 bytes)
`int64_t`	`uint64_t`	64 bits (8 bytes)

x

Example: Limits of Some Types on 32-bit ARM

type	Smallest	largest
`unsigned char`	0	255
`char (ARM – unsigned)`	0	256
`char (INTEL – signed)`	-128	127
`unsigned short`	0	65,535
`short`	-32,768	32,767
`uint32_t`	0	2^32 -1
`int32_t`	-2^31	2^31 -1

- WARNING: If you want to force a char to be signed or unsigned use either
- signed char or unsigned char

x

Watch out for Hardware differences: Example: Is a Char signed or unsigned?

```
#include <stdio.h>
#include <stdlib.h>

int
main(void)
{
    char c = 255;

    printf("%d\n", (int)c);

    return EXIT_SUCCESS;
}
```

- variable c is being cast promoted to an int
- So, what is printed?
- Depends on the hardware
- Aarch32 and Aarch64 (arm) it is unsigned

  255

- Intel 64-bit it is signed

  -1

x

sizeof(): Variable Size (number of bytes) *Operator*

```
#include <stddef.h>
/* size_t type may vary by system but is always unsigned */
```

sizeof() operator returns:

the number of bytes used to store a variable or variable type

```
size_t size = sizeof(variable_type);
        or
size_t size = sizeof(variable_name); // preferred!
```

- The argument to `sizeof()` is often an expression:

```
size = sizeof(int * 10);
```

  - reads as:
    - number of bytes required to store **10 integers (an array of [10])**

x

sizeof() Examples (On the pi-cluster)

```
#include <stdio.h>
#include <stdlib.h>
int
main (void){

    printf("char  is %u\n", sizeof(char));
    printf("short is %u\n",sizeof(short));
    printf("int   is %u\n", sizeof(int));
    printf("long  is %u\n", sizeof(long));
    return EXIT_SUCCESS;
}
```

On 64-bit machines the %u should be %ul

```
% ./a.out (on the picluster)
char  is 1
short is 2
int   is 4
long  is 4
```

Unsigned Integers (positive numbers) with a Fixed # of Bits

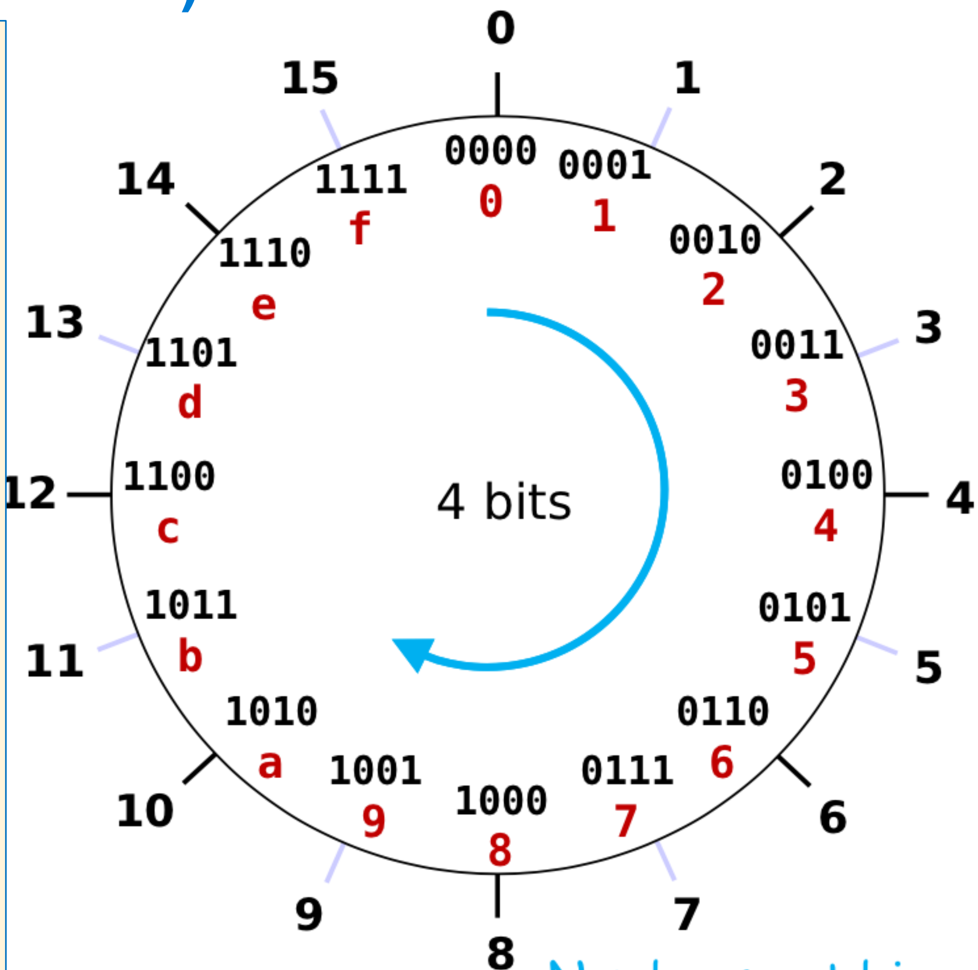
- Example 4 bits is 2^4 = 16 distinct values
- **Modular** (C operator: %) or clock math
  - Numbers start at 0 and "wrap around" after 15 and go back to 0
- Keep adding 1

  wraps (clockwise)

  0000 -> 0001 … -> 1111 -> 0000
- Keep subtracting 1

  wraps (counter-clockwise)

  1111 -> 1110 … -> 0000 -> 1111
- Addition and subtraction use normal "carry" and "borrow" rules, just operate in binary



Numbers get bigger in this direction

x

Addition

- Add two numbers in decimal

+1 carry

$$
\begin{array}{cc}
 & 2\ 7 \\
+ & \underline{2\ 0} \\
 & 4\ 7
\end{array}
\qquad\qquad
\begin{array}{cc}
 & 1\ 2 \\
+ & \underline{7\ 8} \\
 & 9\ 0
\end{array}
$$

- Adding Single digits in binary
  - 0 + 0 = 0
  - 0 + 1 = 1
  - 1 + 0 = 1
  - 1 + 1 = 0 (with 1 carry) or 10 $_{\text{base 2}}$

x

Unsigned Binary Number: Addition in FIXED 8 bits

Be Aware in Binary

1 + 1 = 10 base 10: (1 + 1 = 2)

1 + 1 + 1 = 11 base10: (1 + 1 + 1 = 3)

Carry Bit

carries	0	0	1	0	0	0	1	1		
		1	0	1	0	0	0	0	1	161
+		0	0	1	1	0	0	1	1	51
sum		1	1	0	1	0	1	0	0	212

x

Subtraction

- Add two numbers in decimal

					-1 borrow	+10 borrow
	2	7			6	2
–	1	0		+	2	3
	1	7			3	9

- Subtracting Single digits in binary
  - 0 - 0 = 0
  - 0 - 1 = 1 (with a +2 (a 1) from the column to the left)
  - 1 - 0 = 1
  - 1 - 1 = 0

x

Unsigned Binary Number: Subtraction in **FIXED** 8 bits

borrows

	1	0	1	0	0	0	0	1		161
-	0	0	1	1	0	0	1	1	-	51

difference

Be Aware in Binary

1 - 1 = 0

10 - 1 = 1 base 10: (2 – 1 = 1)

x

Unsigned Binary Number: Subtraction in **FIXED** 8 bits
Build of previous slide – note warning

borrows		10	10	10	10	10	10			
	0	1	0	1	1	1	0	1		161
-	0	0	1	1	0	0	1	1	-	51
difference	0	1	1	0	1	1	1	0		110

Be Aware in Binary

1 - 1 = 0

10 - 1 = 1 base 10: (2 – 1 = 1)

NOTICE

This slide is performing the Subtraction shown in the previous slide using PowerPoint builds when this slide is viewed in a pdf it will look incorrect

x

Unsigned Binary Multiplication example

```
  a = 0 1 0 1 0 0 1 1   (83)
x b = 0 0 0 0 0 0 1 0   (2)
    ---------------
      0 0 0 0 0 0 0 0
    0|1 0 1 0 0 1 1
    ---------------
    0|1 0 1 0 0 1 1 0 (166)
```

Since we are multiplying by a power of two we can shift left by 2 (zero insert at LSB)

```
a = 0 1 0 1 0 0 1 1   (83)

  0 1 0 1 0 0 1 1 0 (166)
```

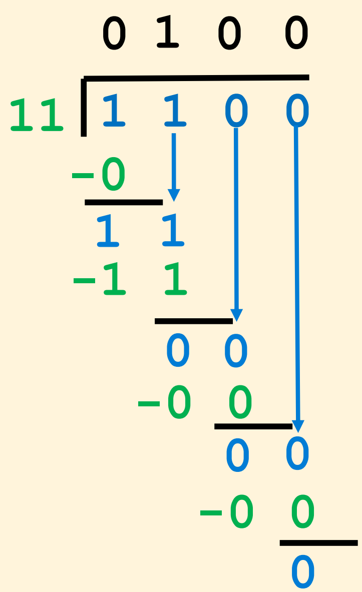
```
  a = 0 1 0 1 0 0 1 1   (83)
x b = 0 0 0 0 0 1 0 1   (5)
    ---------------
      0 1 0 1 0 0 1 1
    0|0 0 0 0 0 0 0
  0 1|0 1 0 0 1 1
  ---------------
  0 1|1 0 0 1 1 1 1 1 (415!!)
```

Exceeds what we can represent in 8 bits (255)

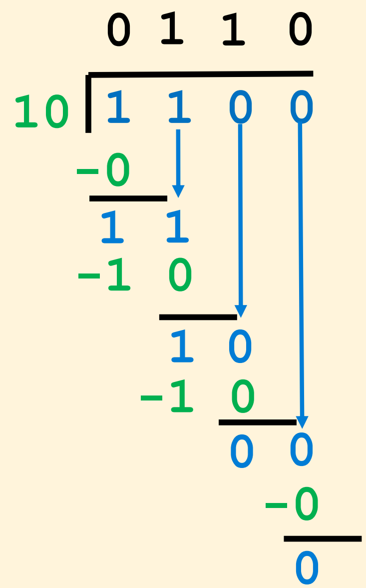
x

Unsigned Binary Number Divide

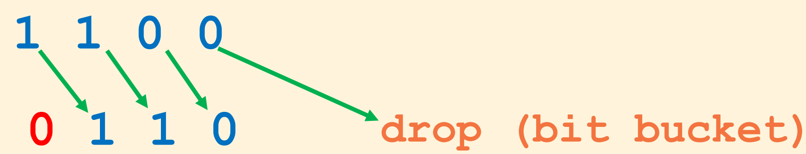
Consider 12/3 = 4



Consider 12/2 = 6



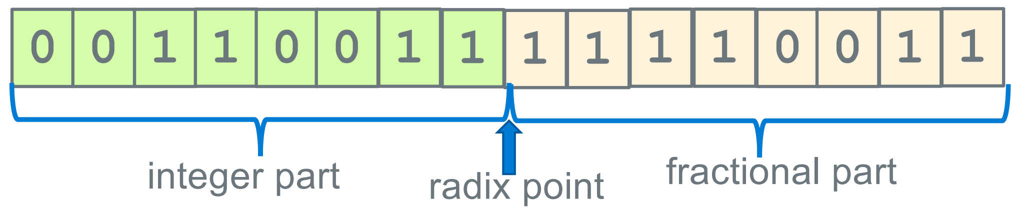
Since we are dividing by a power of two
we can shift right by 1 (zero insert at MSB)



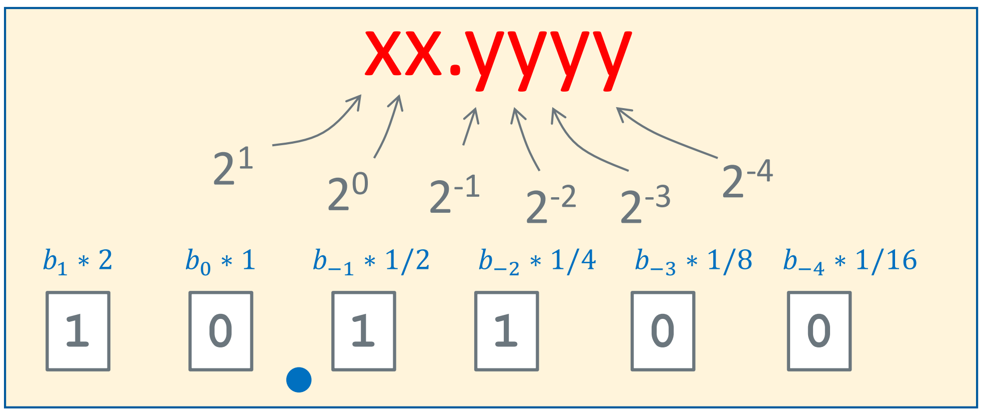
x

Fractional Binary Numbers

Binary	Decimal
2^{-1}	0.5
2^{-2}	0.25
2^{-3}	0.125
2^{-4}	0.0625



- ***"Binary Point,"*** like ***decimal point***, signifies boundary between integer and fractional parts
- Bits to right of "binary point" represent fractional powers of 2
- Example: $10.1010_2 = 1\times2^1 + 1\times2^{-1} + 1\times2^{-3} = 2.625_{10}$



x

Fixed Point Binary Number Divide

Consider 10/4 = 2.5

```
       0 0 1 0 .1
     ____________
 100 |1 0 1 0 .0
     -1 0 0
     ------
        0 1 0
      -0 0 0
      -------
          1 0   0
        -1 0   0
       ----------
                0
```

Consider 10/8 = 1.25

```
        0 0 0 1 .0 1
      ______________
 1000 |1 0 1 0 .0 0
      -1 0 0 0
      --------
         0 1 0  0
       -0 0 0 0
       ----------
            1 0 0 0
          -1 0 0 0
        --------------
                0
```


x

Fixed Point Binary Number Limitations

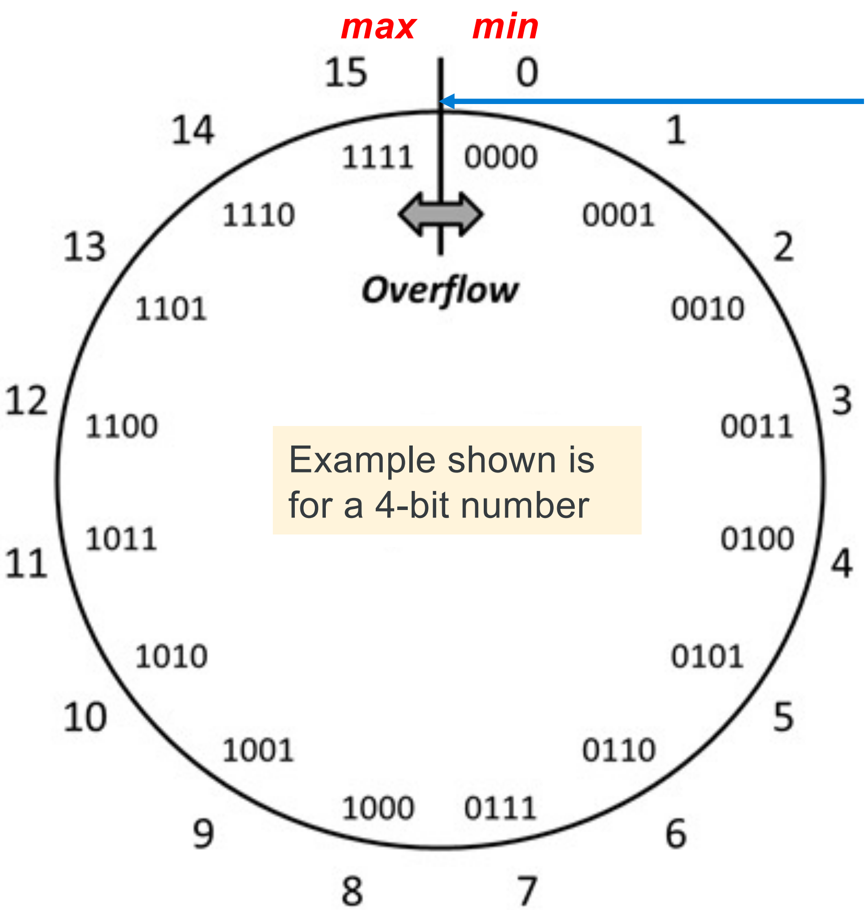
- Limitation #1
  - Can only exactly represent numbers of the form $x/2^k$
    - Other rational numbers have repeating bit representations

Value	Representation
1/3	`0.0101010101[01]`...$_2$
1/5	`0.001100110011[0011]`...$_2$
1/10	`0.0001100110011[0011]`...$_2$

- Limitation #2
  - Just one setting of binary point within the *w* bits
  - Limited range of numbers (very small values? very large?)

x

Overflow: Going Past the Boundary Between max and min



Overflow: Occurs when an arithmetic result (from addition or subtraction for example) is is more than **min** or **max** limits

C (and Java) ignore overflow exceptions

- You end up with a bad value in your program and absolutely no warning or indication… happy debugging!….

x

Unsigned Integer Number Overflow: Addition in 8 bits

	Carry Bit									
carries	1	1	1	1	1	1	1	1		
		1	0	1	0	0	0	0	1	161
+		0	1	0	1	1	1	1	1	95
sum		0	0	0	0	0	0	0	0	256

only 8 bits for numbers in this example carry bit is always dropped from result

Rule: When Carry Bit != 0, overflow has occurred for unsigned integers!

x

Unsigned Integers (positive numbers) with a Fixed # of Bits

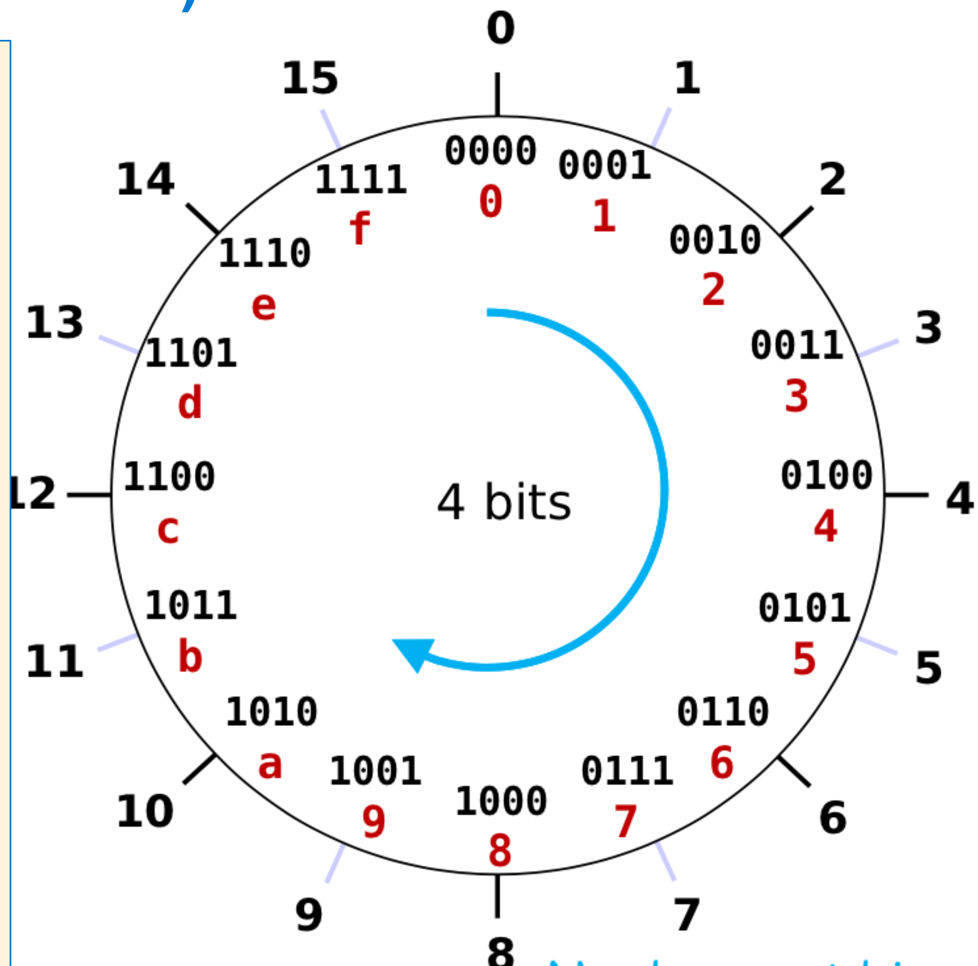
- Example 4 bits is 2^4 = 16 distinct values
- **Modular** (C operator: %) or clock math
  - Numbers start at 0 and "wrap around" after 15 and go back to 0
- Keep adding 1

  wraps (clockwise)

  0000 -> 0001 … -> 1111 -> 0000
- Keep subtracting 1

  wraps (counter-clockwise)

  1111 -> 1110 … -> 0000 -> 1111
- Addition and subtraction use normal "carry" and "borrow" rules



Numbers get bigger in this direction

x

Overflow: Unsigned Values 4-bit limit

Overflow: Occurs when an arithmetic result is **exceeds** the min or max limits

Addition Overflow: hardware drops carry

```
  15
+  2
----
  17
```

only 4 bits for numbers in this example

carry bit is always dropped from result

```
  1111
+ 0010
------
 10001
oops 1
```

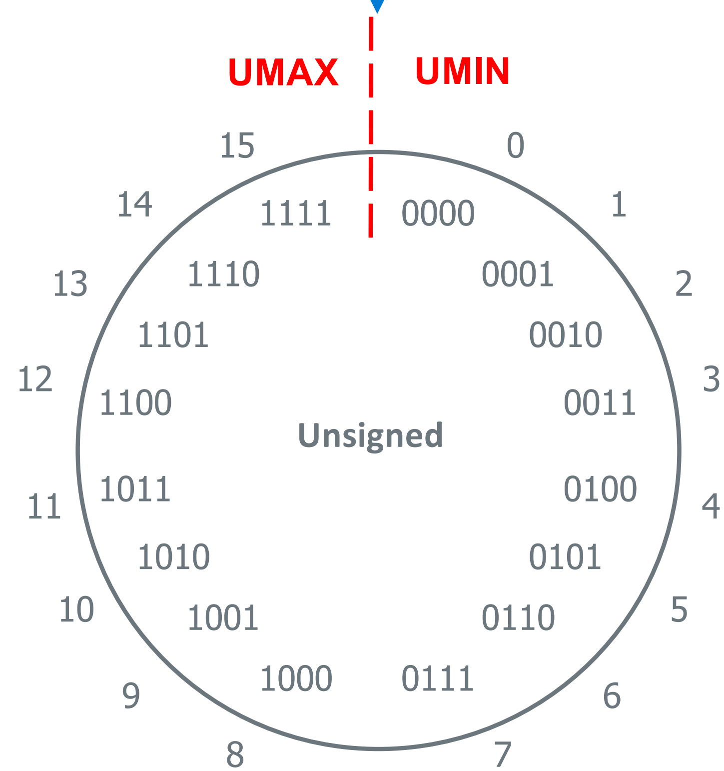
Subtraction Overflow: drops the borrow

```
   1
-  2
----
  -1
```

only 4 bits for numbers in this example

carry bit is always dropped from result

```
 10001
-  0010
-------
  1111
oops 15
```



x

Problem: How to Encode Both Positive and Negative Integers

- How do we represent the negative numbers within a fixed number of bits?
  - Allocate some bit patterns to negative and others to positive numbers (and zero)
- $\mathbf{2^n}$ distinct bit patterns to encode positive and negative values
- **Unsigned values:** $0 \ldots 2^n\text{–}1$ ← -1 comes from counting 0 as a "positive" number
- **Signed values:** $-2^{n-1} \ldots 2^{n-1}\text{–}1$ (dividing the range in ~ half including 0)
- **On a number line (below):** 8-bit integers – signed and unsigned (*e.g.*, **`char in C`**)

Same “width” (same number of encodings), just shifted in value

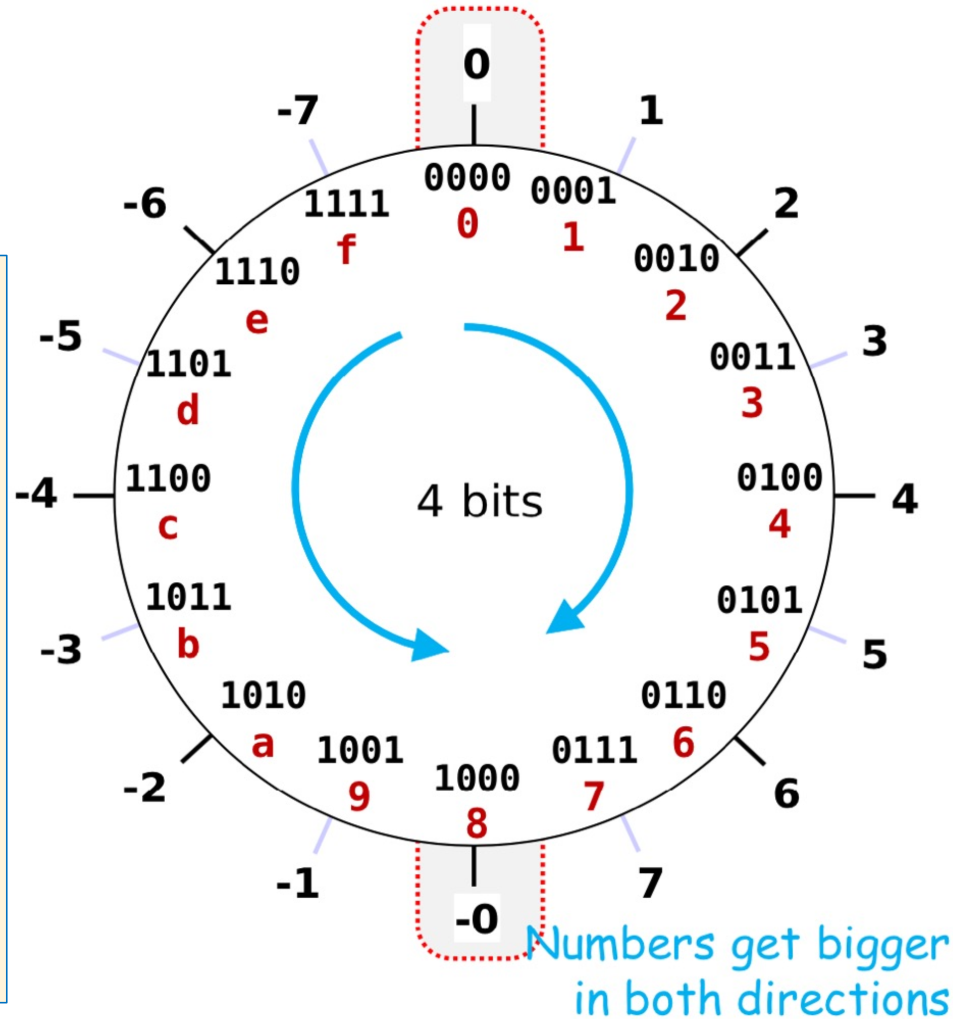
x

Negative Integer Numbers: Sign + Magnitude Method

- Use the Most Significant Bit as a sign bit
  - 0 as the MSB represents positive numbers
  - 1 as the MSB represents negative numbers
- Two (oops) representations for zero: 0000, 1000
- Tricky Math (must handle sign bit independently)

4	0100	4	0100
− 3	− 0011	+ −3	+ 1011
1	0001	−7	1111
	✓		✗

- With Simple math, Positive and Negatives "*increment*" (+1) in the opposite directions!

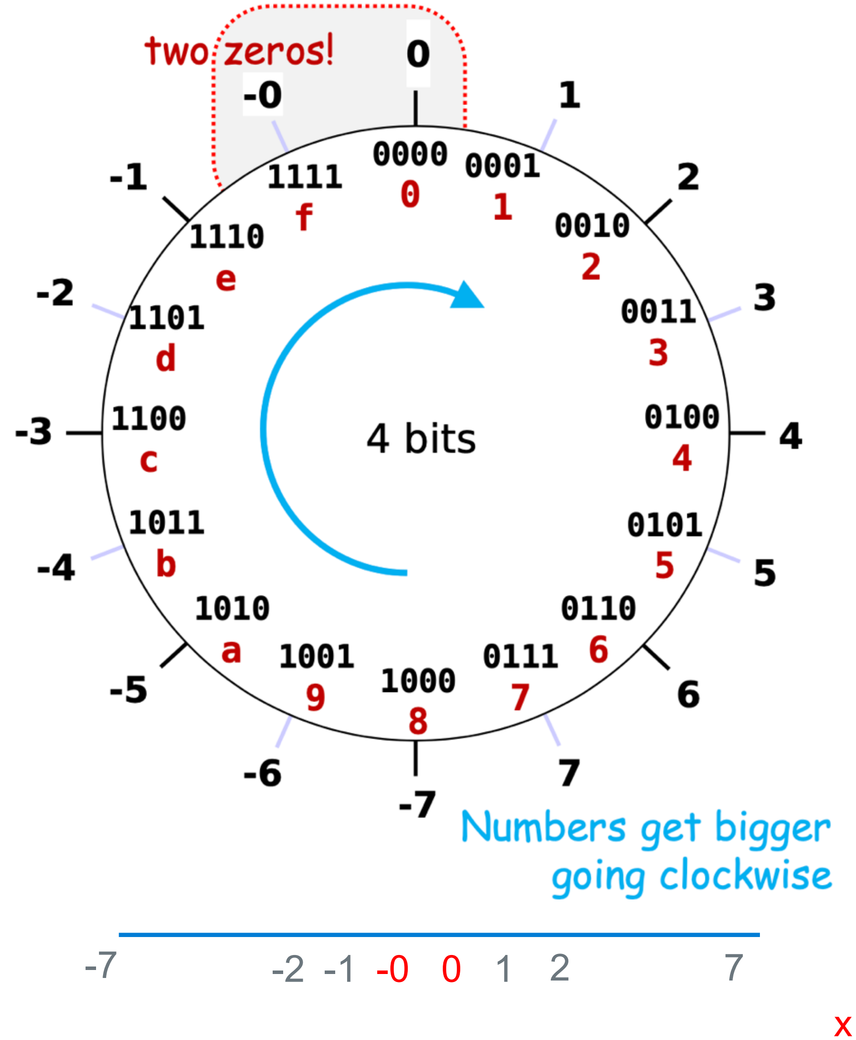


1's Complement Signed Integer Method

- Use the MSB is the sign bit to represent a negative value is encoded as the 1's complement
- All negative values have a one in the leftmost bit
- All positive values have a zero in the leftmost bit

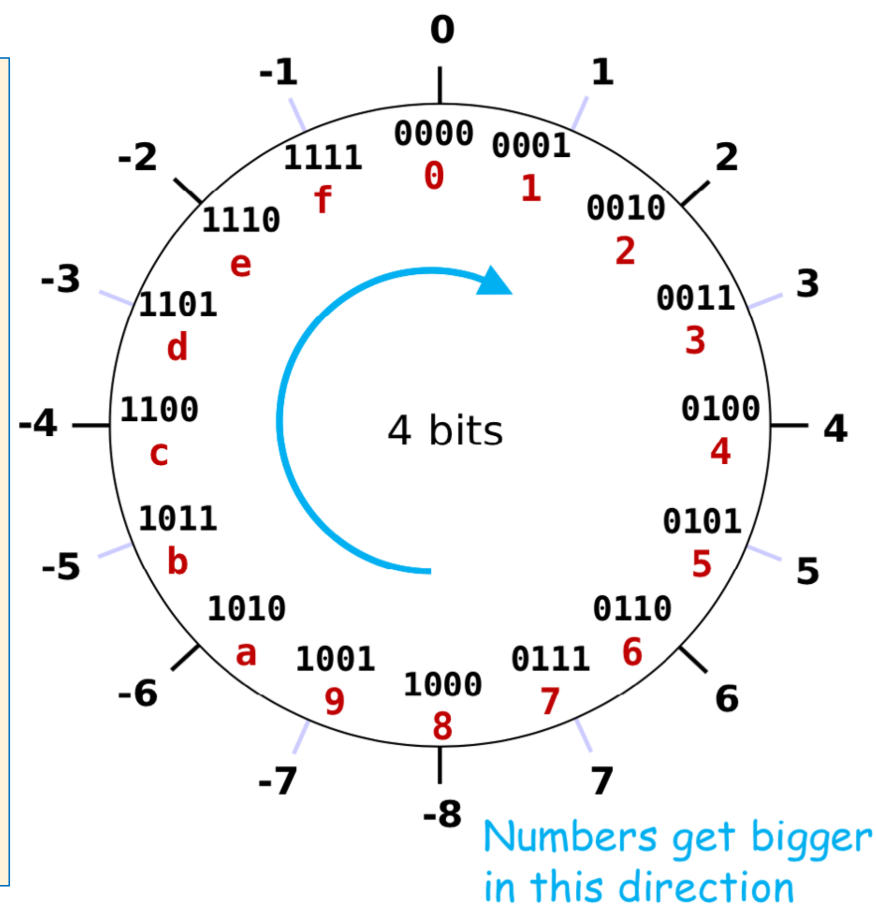
Number	+	-
0	0000	1111
1	0001	1110
2	0010	1101
3	0011	1100
4	0100	1011
5	0101	1010
6	0110	1001
7	0111	1000

- The problem is **there are two values for zero**
  - 1111 and 0000 in 4-bits
  - arithmetic is tricky when you cross over the zeros



2's Complement Signed Integer Method

- Positive numbers encoded same as unsigned numbers
- All negative values have a one in the leftmost bit
- All positive values have a zero in the leftmost bit
  - This implies that 0 is a positive value
- Only one zero
- **For n bits, Number range is $-(2^{n-1})$ to $+(2^{n-1}$ - 1)**
  - Negative values "go further" than the positive values
- Example: the range for 8 bits:
  -128, -127, .. 0, .. 126, **+127**
- Example the range for 32 bits:
  -2147483648 .. 0, .. **+2147483647**
- *Arithmetic is the same as with unsigned binary!*



x

Two's Complement: The MSB Has a *Negative Weight*

$$2's\ Comp = -b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \ldots + b_1 2^1 + b_0 2^0$$

b_{n-1} weight is (-2^{n-1}), all other bits: have positive weights $(+2^i)$

b_{n-1}	b_{n-2}	. . .	b_0

- 4-bit (w = 4) weight = -2^{4-1} = -2^3 = -8
  - 1010_2 **unsigned**:
    $1\text{x}2^3 + 0\text{x}2^2 + 1\text{x}2^1 + 0\text{x}2^0 = \mathbf{10}$
  - 1010_2 **two's complement**:
    $-1\text{x}2^3 + 0\text{x}2^2 + 1\text{x}2^1 + 0\text{x}2^0 = -8 + 2 = \mathbf{-6}$
  - -8 in **two's complement:**
    $1000_2 = -2^3 + 0 = -8$
  - -1 in **two's complement:**
    $1111_2 = -2^3 + (2^3 - 1) = -8 + 7 = \mathbf{-1}$

Summary: Min, Max Values: Unsigned and Two's Complement

Two's Complement → Unsigned for n bits

- **Unsigned Value Range**

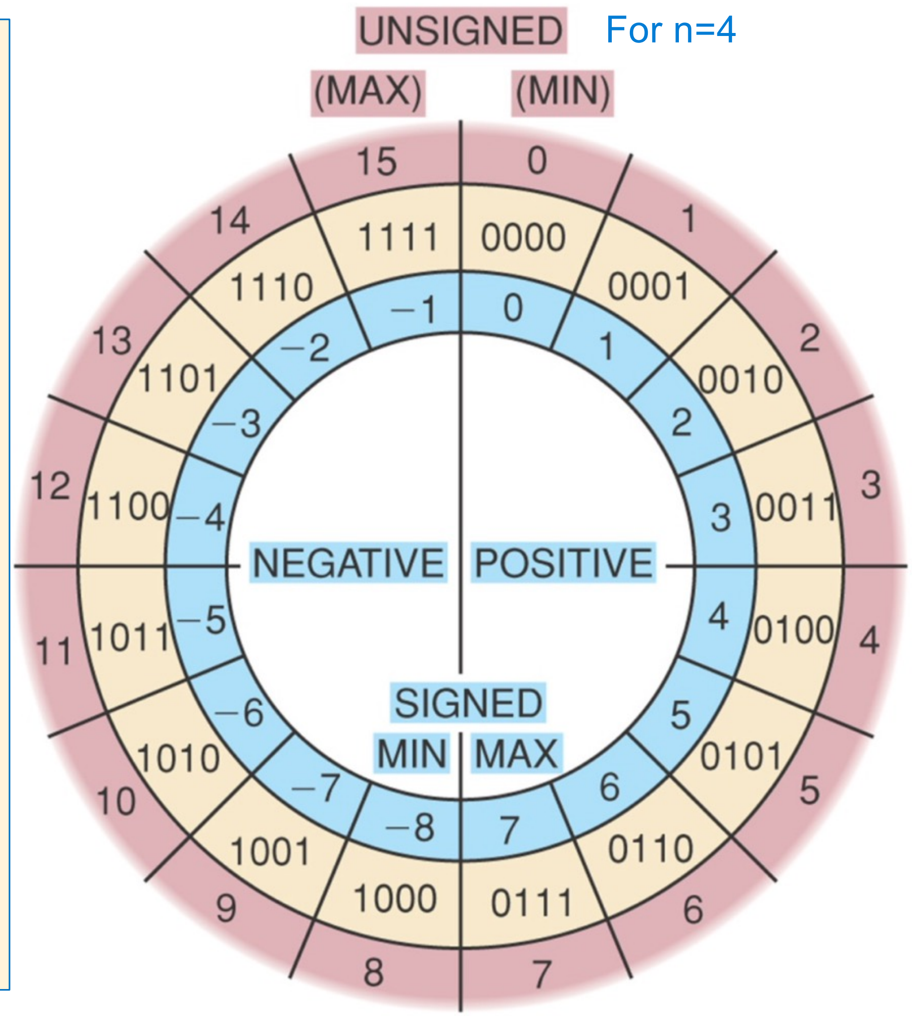
UMin = `0b00…00`
= $\mathbf{0}$

UMax = `0b11…11`
= $\mathbf{2^n - 1}$

- **Two's Complement Range**

SMin = `0b10…00`
= $\mathbf{-2^{n-1}}$

SMax = `0b01…11`
= $\mathbf{2^{n-1} - 1}$



x

Negation Of a Two's Complement Number (Method 1)

```
-x == ~x + 1;
```

```
       7 = 0111
invert = 1000
add 1   +    1
      -7   1001
```

```
      -7 = 1001
invert = 0110
add 1   +    1
       7   0111
```

```
 7 =          0111
-7 =      +   1001
(discard carry) 0000
```

```
       1 = 0001
invert = 1110
add 1   +    1
      -1   1111
```

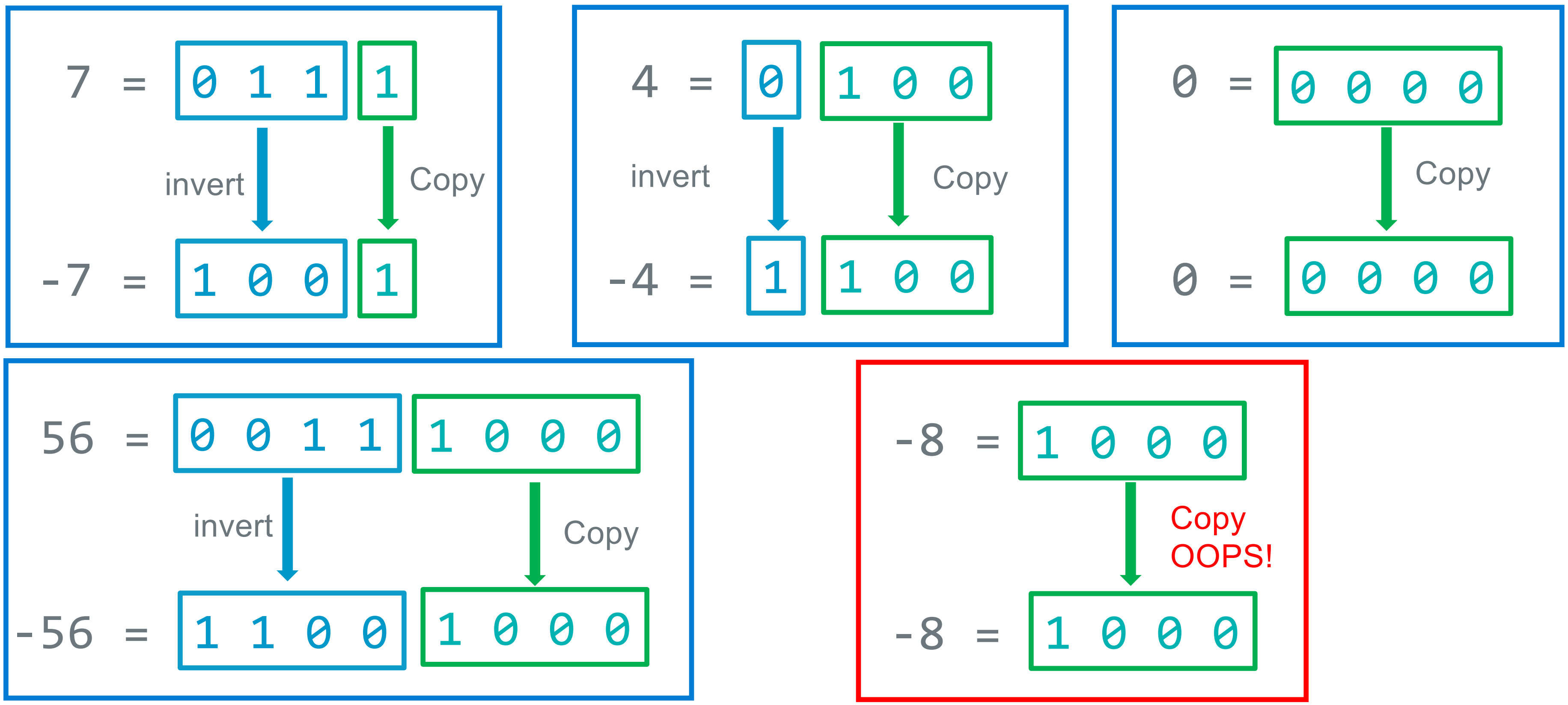
```
      -1 = 1111
invert = 0000
add 1   +    1
       1   0001
```

```
      -8 = 1000
invert = 0111
add 1   +    1
      -8   1000  oops!
```


x

Negation of a Two's Complement Number (Method 2)

1. **copy unchanged** right most bit containing a 1 and all the 0's to its right
2. Invert all the bits to the left of the right-most 1



x

Signed Decimal to Two's Complement Conversion

dividend -102	Quotient	Remainder	Bit Position
102/2	51	0	b0
51/2	25	1	b1
25/2	12	1	b2
12/2	6	0	b3
6/2	3	0	b4
3/2	1	1	b5
1/2	0	1	b6
0/2	0	0	b7

102(base 10) = $b_7\,b_6\,b_5\,b_4\,b_3\,b_2\,b_1 b_0 \;=\; 0b0110\;0110$

Get the two complement of 01100110 is 10011010

x

Two's Complement to Signed Decimal Conversion - Positive

What is $\overset{b_7}{0}\ \overset{b_6}{1}\ \overset{b_5}{1}\ \overset{b_4}{0}\ \overset{b_3}{0}\ \overset{b_2}{1}\ \overset{b_1}{0}\ \overset{b_0}{1}$ (base 2) in decimal (N)?

Signed Bit Bias	Bit	Bit Position	Bias
$-2^{W-1} = -2^{8-1}$ = -128	x 0	b7	0
Product Shift Left	Addend	Bit Position	Product
2 x 0 = 0 (shift left)	+ 1	b6	1
2 x 1 = 2	+ 1	b5	3
2 x 3 = 6	+ 0	b4	6
2 x 6 = 12	+ 0	b3	12
2 x 12 = 24	+ 1	b2	25
2 x 25 = 50	+ 0	b1	50
2 x 50 = 100	+ 1	b0	SUM = 101
		Bias + SUM:	0 + 101 = 101

x

Two's Complement to Signed Decimal Conversion - Negative

What is $\overset{b_7}{1}\ \overset{b_6}{1}\ \overset{b_5}{1}\ \overset{b_4}{0}\ \overset{b_3}{0}\ \overset{b_2}{1}\ \overset{b_1}{0}\ \overset{b_0}{1}_{(\text{base } 2)}$ in decimal (N)?

Signed Bit Bias	Bit	Bit Position	Bias
$-2^{W-1} = -2^{8-1} = -128$	x 1	b7	-128
Product Shift Left	Addend	Bit Position	Product
2 x 0 = 0 (shift left)	+ 1	b6	1
2 x 1 = 2	+ 1	b5	3
2 x 3 = 6	+ 0	b4	6
2 x 6 = 12	+ 0	b3	12
2 x 12 = 24	+ 1	b2	25
2 x 25 = 50	+ 0	b1	50
2 x 50 = 100	+ 1	b0	SUM = 101
		Bias + SUM:	-128 + 101 = **-27**

x

Two’s Complement Addition and Subtraction

- **Addition:** just add the two number directly
- **Subtraction:** you can convert to addition: **difference = minuend – subtraction**
  difference = minuend + 2’s complement (subtrahend)

```
      Cout
        0   0   0   0   0   0   1   1
      x =   0   1   0   1   0   0   1   1
      y =   0   0   0   0   1   0   1   1
          -------------------------------
  x + y =   0   1   0   1   1   1   1   0
```

```
    x =  0  1  0  1  0  0  1  1
    y =  0  0  0  0  1  0  1  1
        ------------------------
x-y   =  0  1  0  0  1  0  0  0
```

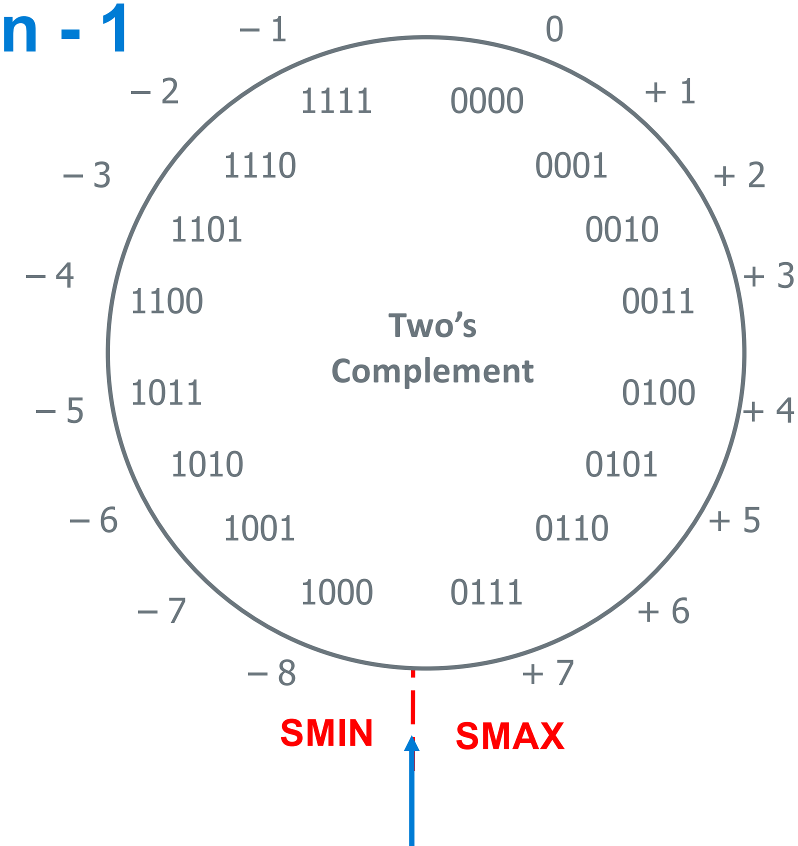
```
                  2’s complement first and then add
                      x  =  0  1  0  1  0  0  1  1
                 + (-y)  =  1  1  1  1  0  1  0  1
                           ------------------------
x - y = x +(-y)          =  0  1  0  0  1  0  0  0
```


x

Two’s Complement Overflow Detection - 1

- When adding two positive numbers or two negative numbers
- **4-bit** Two’s complement numbers (positive overflow)

Cout	Cin					
0	1	0	0			
	0	1	0	1		5
+	0	1	1	0		6
	1	0	1	1	-5	!= 11



Overflow: Occurs when an arithmetic result is beyond the min or max limits

x

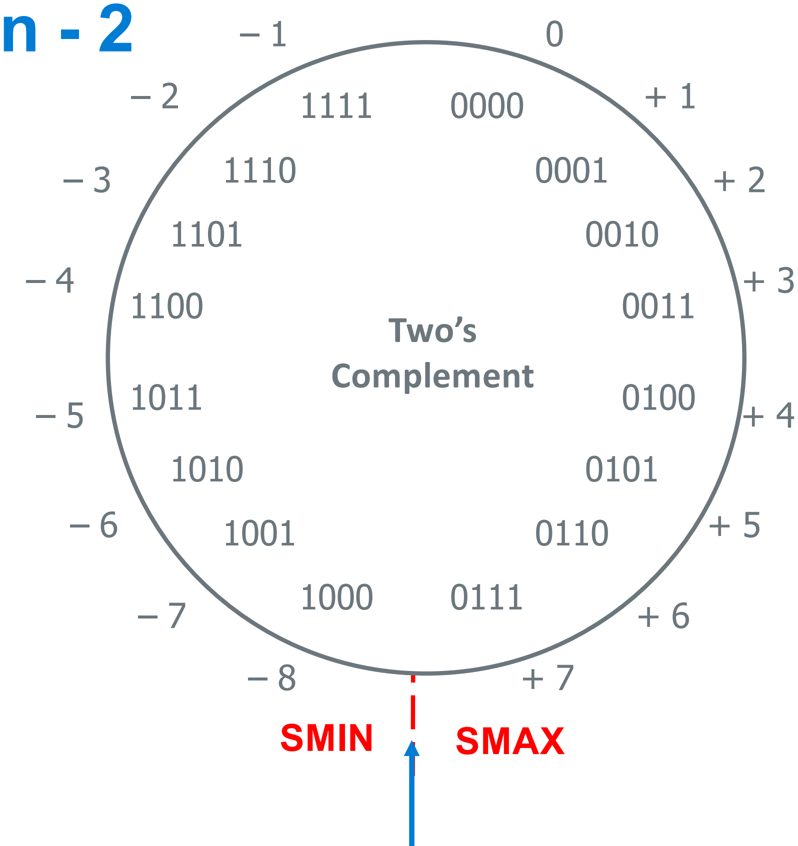
Two's Complement Overflow Detection - 2

- When adding two positive numbers or two negative numbers
- **4-bit** Two's complement numbers (negative overflow)

carry bit is dropped from result

```
Cout Cin
1    0   1   1
     1   0   0   1        -7
  +  1   0   1   1        -5
  ----------------------------
     0   1   0   0   +4   != -12
```

signed numbers: overflow occurs if
operands have same sign and result's sign is different



Overflow: Occurs when an arithmetic result is beyond the min or max limits

x

Summary: When Does Overflow Occur

Operand 1
+ Operand 2
Result

Operand 1 Sign	Operand 2 Sign	Is overflow Possible?
+	+	**YES**
–	–	**YES**
+	–	**NO**
–	+	**NO**

x

Scientific Notation Binary

exponent

$$+\overbrace{1.01}^{\text{mantissa}}{}_{2} \times 2^{-1}$$

sign

binary point

radix (base)

- Computer hardware that supports this is called floating point hardware due to the “floating” of the binary point
- Declare such variable in C as **`float`** (or **`double`**)

x

Floating Point Representation

- Analogous to scientific notation
- In Decimal:
  - Not 12000000, but 1.2×10^7 In C: 1.2e7
  - Not 0.0000012, but 1.2×10^{-6} In C: 1.2e-6
- In Binary:
  - Not 11000.000, but 1.1×2^4
  - Not 0.000101, but 1.01×2^{-4}

x

Normalized Scientific Notation

- Convert from scientific notation to fixed binary point
- Perform the multiplication by shifting the decimal until the exponent disappears

Binary	Decimal
2^{-1}	0.5
2^{-2}	0.25
2^{-3}	0.125
2^{-4}	0.0625

  - Example: $1.011_2 \times 2^4 = 10110_2 = 22_{10}$
  - Example: $1.011_2 \times 2^{-2} = 0.01011_2 = 0.34375_{10}$
- Convert from binary point to *normalized* scientific notation
  - Distribute out exponents until binary point is to the right of a single digit
  - Example: $1101.001_2 = 1.101001_2 \times 2^3$

x

Encoding Fractions Observations

In Base 2:

$10.1 \times 2^5 = 1.01 \times 2^6$

$1011.1 \times 2^5 = 1.0111 \times 2^8$

$0.110 \times 2^5 = 1.10 \times 2^4$

Normalizing with base 2 :
adjust so there *always* a 1 to the **left of the decimal point**!
this 1 is **called the hidden bit** as we do not have use a bit to store it since it is there in every normalized mantissa

- Adjust x to always be in the format **1.XXXXXXXXX… (fraction is normalized)**
- Fraction portion ONLY **encodes** what is *to the* ***right*** of the decimal point
- "Hidden bit" allows number to have **One additional digit for <u>increased</u> precision**

Fraction encoding is 1.[FRACTION BINARY DIGITS]

x

Floating Point Numbers: Implementation Approach

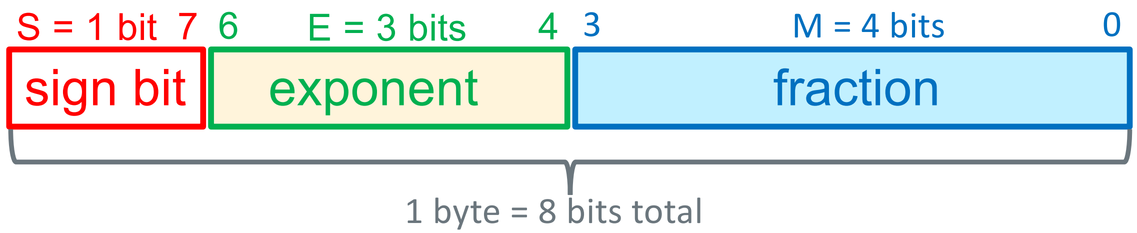
- Supports a wide range of numbers
- Flexible "floating" decimal point
- Represent scientific notation numbers like $1.202 \text{ x } 10^6$

$$(-1)^s \, M \;\; 2^E$$

S	E	M
sign bit	exponent	fraction

- **Sign bit** (a single bit): 0 positive, 1 negative
- **Exponent**: **encoding** of E above (it is NOT E directly represented in binary)
- **Fraction**: **encoding** of M above (it is NOT M directly represented in binary)

Floating Point Number in a Byte (Not A Real Format)



- **Mantissa encoding: = 1.[xxxx] encoded as an unsigned value**
  - **4 bits = 16 values + leading digit is always a 1**
- **Exponent encoding:** 3 bits encoded to represent both – and + exponents

x

IEE 754 Floating Point

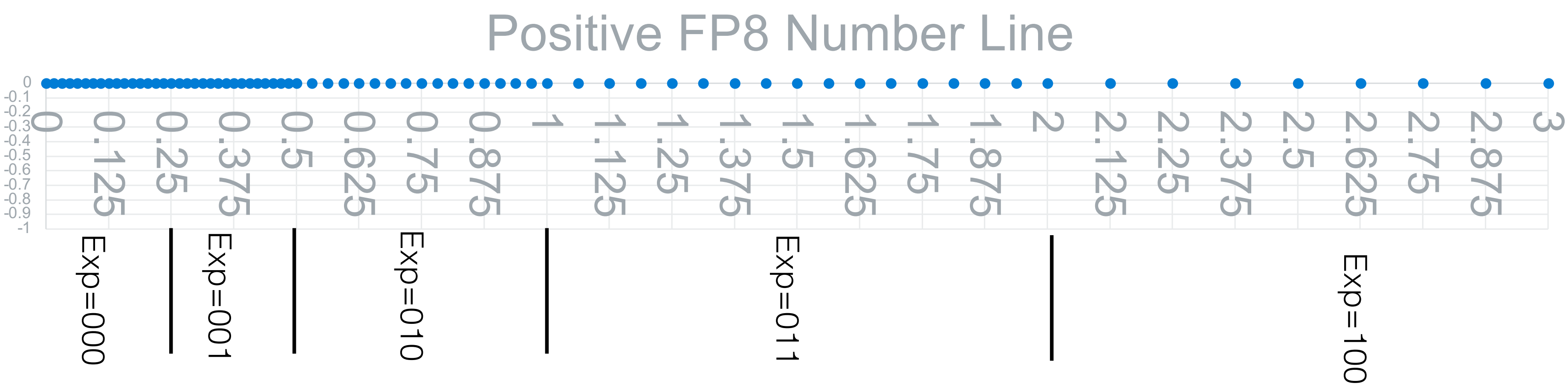
sign	Exp = E +127	mantissa
1	8	23

sign	Exp = E + 1023	mantissa
1	11	52

- Evolving Standard
- **Single – 32 bit "C Float"**
- **Double – 64 bit "C Double"**
- Half – 16 bit
- Quad – 128 bit
- Binary Floating Point
- Standard also defined decimal FP (not supported by most hardware)
- Special Encodings
  - NAN – not a number (quiet, signaling)
  - +∞ and -∞ (biggest positive #, and smallest negative number)
- Subnormal Numbers

x

Floating Point and Linearity



Why the non-linearity?

Note, Exp=000 is treated specially

x

IEEE Floating Point and "C" FP Types

IEEE Type	size	sign	exponent	mantissa	“C” name
bfloat (not IEEE yet)	16	1	8	7	(subset of float)
half	16	1	5	10	
single	32	1	8	23	
double	64	1	11	52	
quad	128	1	15	112	

x

Memory Review: Organized in Units of Bytes

- One bit (digit) of storage (in memory) has two possible **states**: 0 or 1
- Memory is organized into a **fixed unit** of 8 bits, called a **byte**

MSB → Most Significant Bit

Bit position in Byte							
7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	0
Content of each bit							

LSB ← (Least Significant Bit)

- Conceptually, memory is a single, **large array** of **bytes**, **where each byte** has a unique *address (byte addressable memory)*
- An address is an **unsigned** (positive #) *fixed-length* n-bit binary value
  - Range (domain) of possible addresses = *address space*
- Each byte in memory can be **individually accessed** and operated on given its **unique address**

n-bit Memory Address

High address

n-bit Memory Address	Memory contents
..00000111	10101010
..00000110	01010101
..00000101	10101010
..00000100	01010101
..00000011	10101010
..00000010	01010101
..00000001	10101010
..00000000	01010101

Low address

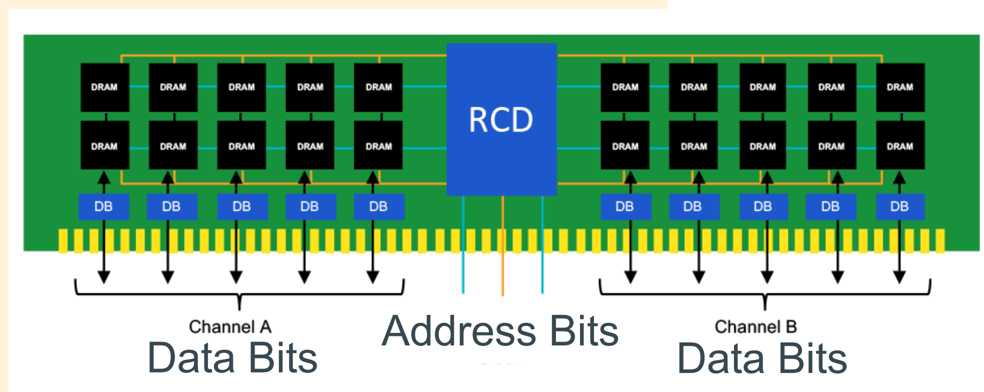
1 byte (8-bits wide)

x

Memory Size

- **Since memory addresses are implemented in hardware using binary**
  - The **Size (number of byte sized cells)** of Memory is specified in **powers of 2**
- Memory size/capacity in **bytes** is specified by the "Number of bits" in an address
  - 32 bits of address = 2^{32} = 4,294,967,296
  - Address Range is 0 to $2^{32} - 1$ (unsigned)
- Shorthand notation for address size (Memory Capacity)
  - KB = 2^{10} (K=1024) kilobyte
  - MB = 2^{20} megabyte
  - GB = 2^{30} gigabyte
  - TB = 2^{40} terabyte
  - PB = 2^{50} petabyte

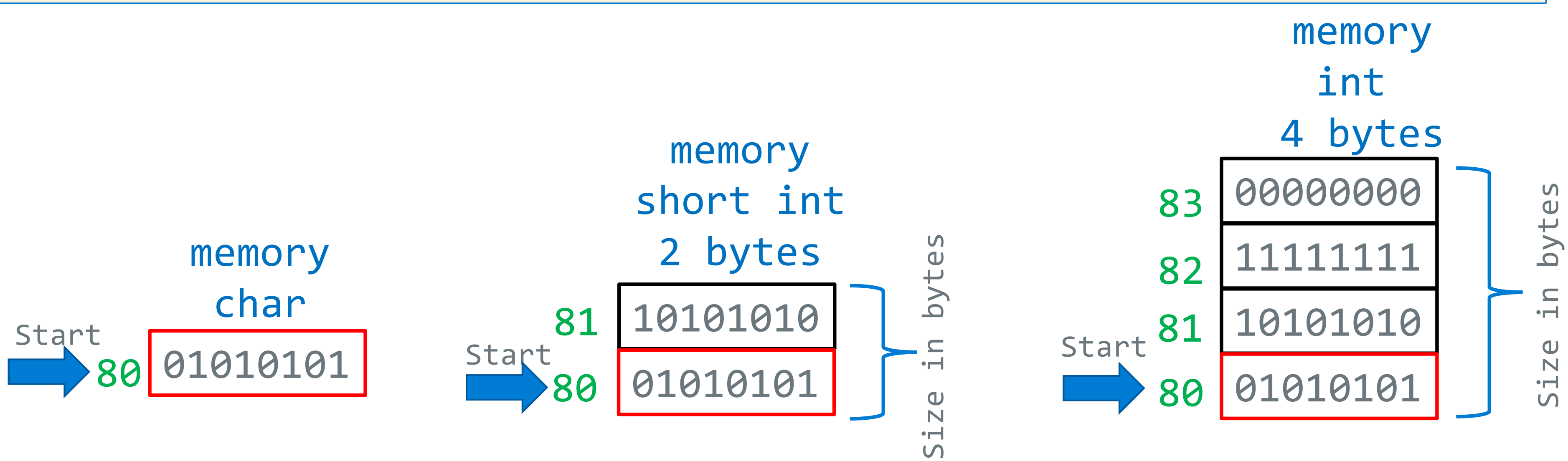
Memory dimm



x

Variables in Memory: Size and Address

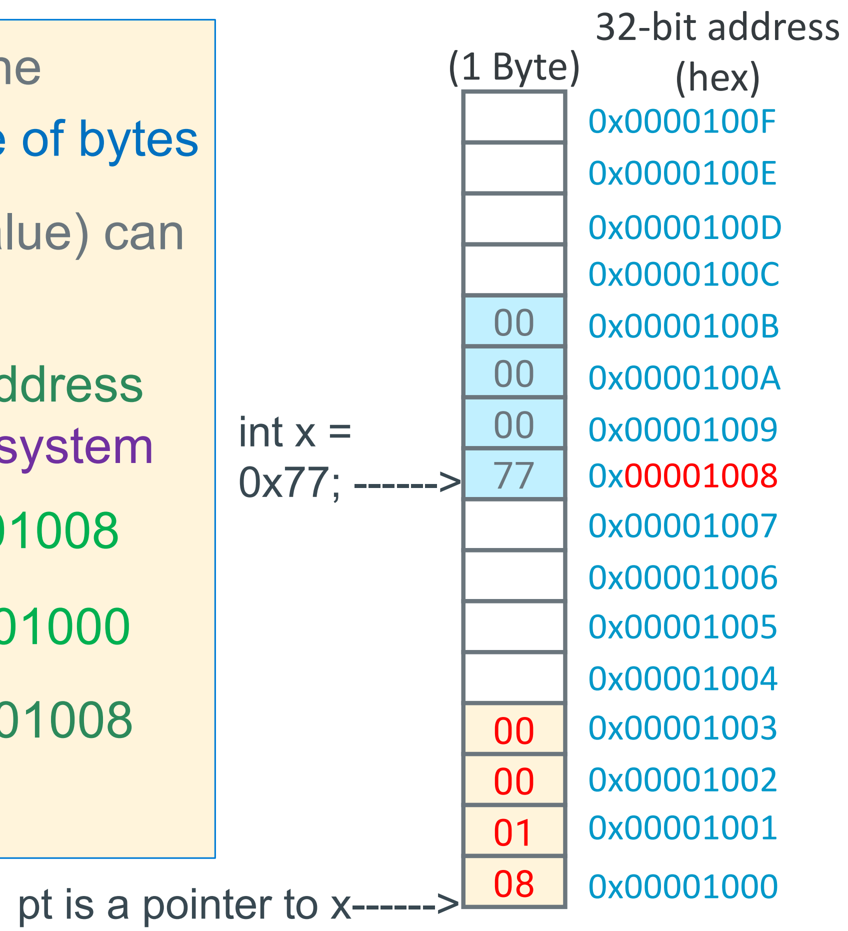
- The number of **contiguous bytes** a variable uses is based on the *type* of the variable
  - Different variable types require different numbers of contiguous bytes
- ***Variable names*** map to a *starting address in memory*
- Example Below: Variables all starting at address 0x80



x

Address and Pointers

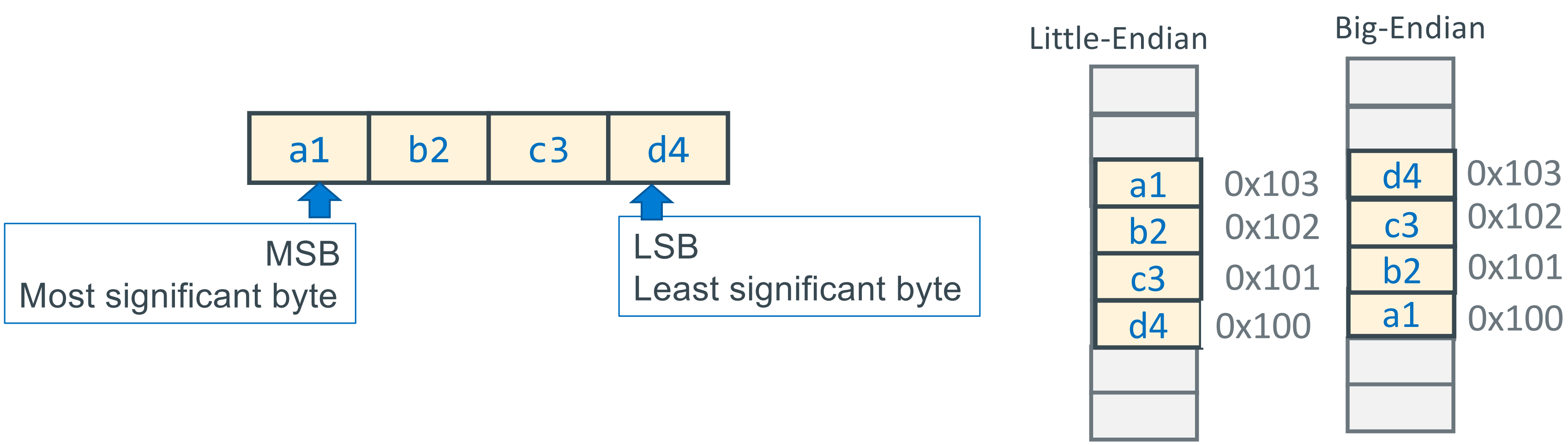
- An address refers to a location in memory, the lowest or first byte in a contiguous sequence of bytes
- A pointer is a variable whose contents (or value) can be properly used as an address
  - The value in a pointer *should* be a valid address allocated to the process by the operating system
- The variable x is at memory address 0x00001008
- The variable pt is at memory location 0x00001000
- The contents of pt is the address of x 0x00001008



x

Byte Ordering of Numbers In Memory: Endianness

- Two different ways to place multi-byte integers in a byte addressable memory
- Big-endian: Most Significant Byte ("big end") starts at the *lowest (starting)* address
- Little-endian: Least Significant Byte ("little end") starts at the *lowest (starting)* address
- Example: 32-bit integer with 4-byte data

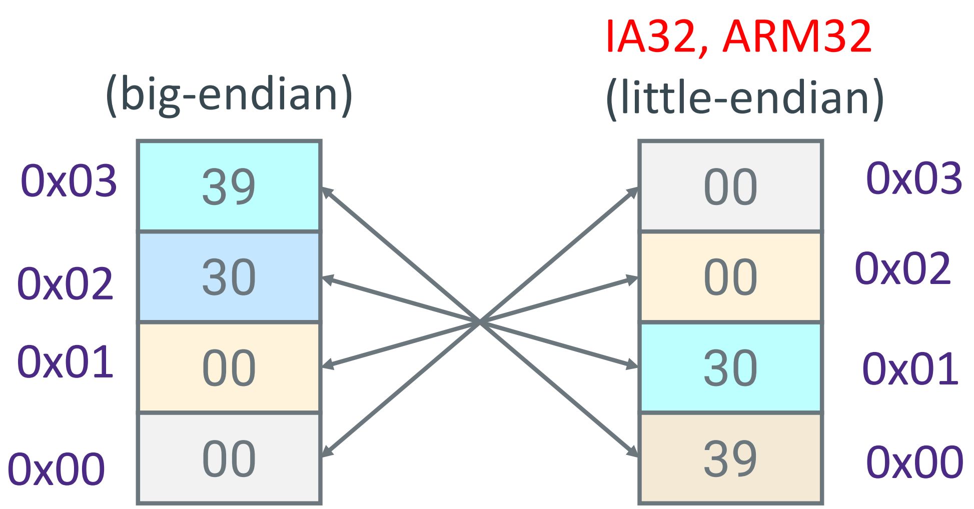


x

Byte Ordering Example

```
Decimal:   12345
Binary:     0011  0000  0011  1001
Hex:           3     0     3     9
```

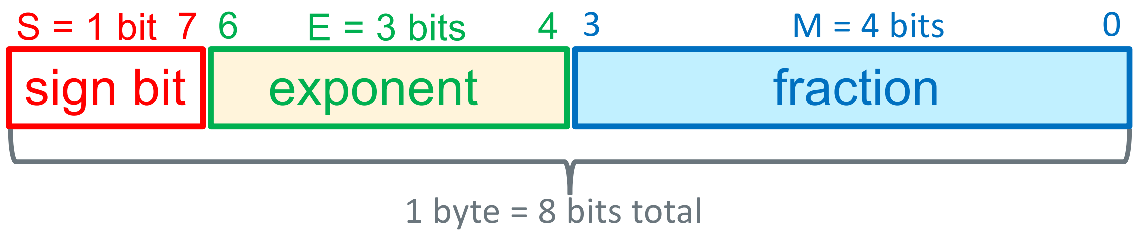
```
int x = 12345;
// or x = 0x3039;
```



x

Extra Slides

Floating Point Number in a Byte (Not A Real Format)



- **Mantissa encoding: = 1.[xxxx] encoded as an unsigned value**
- **Exponent encoding:** 3 bits encoded as an unsigned value using bias encoding
  - **Bias encoding** = **$(2^{E-1} – 1)$**
  - 3 bits for the bias we have $2^{3-1} - 1 = 2^2 - 1$ = a bias of 3
  - **With a Bias of 3**: positive and negative numbers range: small to large is: 2^{-3} to 2^4

Actual	-3	-2	-1	0	1	2	3	4
Bias	**+ 3**	**+ 3**	**+ 3**	**+ 3**	**+ 3**	**+ 3**	**+ 3**	**+3**
Biased	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**

x

Encoding Fractions Observations

Examples In Base 10:

$42.4 \times 10^5 = 4.24 \times 10^6$

$324.5 \times 10^5 = 3.245 \times 10^7$

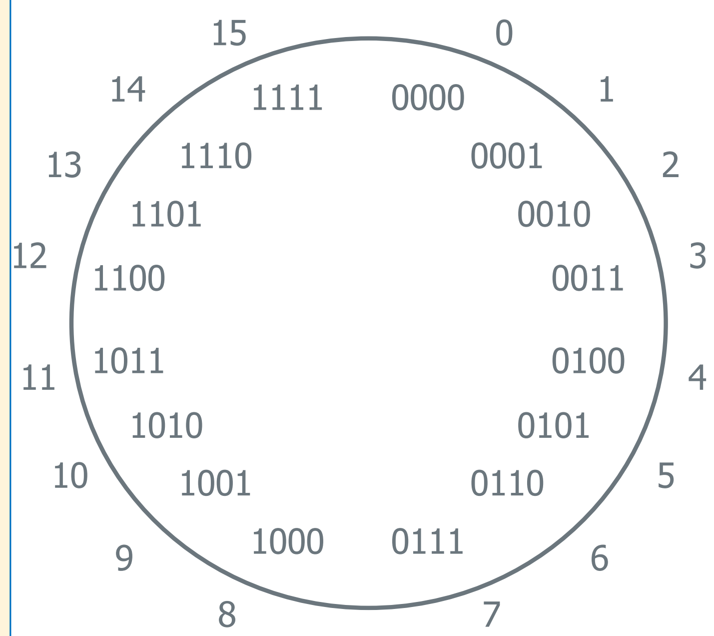
$0.624 \times 10^5 = 6.24 \times 10^4$

Observation on base 10:
We usually adjust the exponent until we get down to one digit to the left of the decimal point

x

Characteristics of Signed Numbers

- Digital Hardware (and C) supports two flavors of integers
  - *unsigned* – only non-negative (positive) numbers
  - *signed* – both negative and non-negatives (positive) numbers
- A Signed integer must be able to represent:
  - Negative integer
  - Zero (0)
  - Positive Integer
  - number + (- representation of number) = 0
- So, with a fixed number of bits, some of the bit patterns in the wheel at the left must be reallocated to represent negative numbers



x

Sign Extension in C: Type casts

- Convert from smaller to larger integral data types
- C and Java automatically performs sign extension
- Example (remember we are working with 32-bit int and 16-bit short)

```
short int sx =   12345;
int       ix = (int) sx;
```

0b0011

Var	Decimal	Hex	Binary
sx	12345	30 39	00110000 00111001
ix	12345	00 00 30 39	00000000 00000000 00110000 00111001

```
short int sy =  -12345;
int       iy = (int) sy;
```

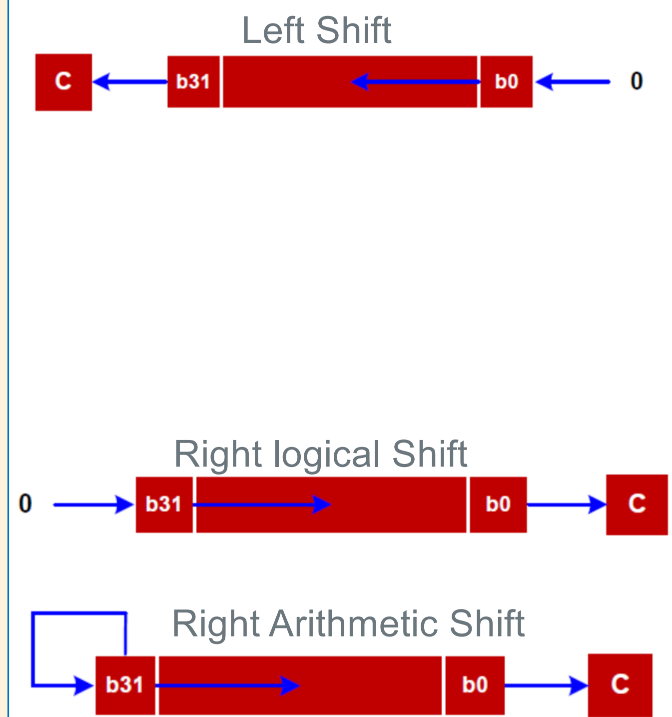
0b1100

Var	Decimal	Hex	Binary
sy	-12345	CF C7	11001111 11000111
iy	-12345	FF FF CF C7	11111111 11111111 11001111 11000111

x

Shift Operations in C

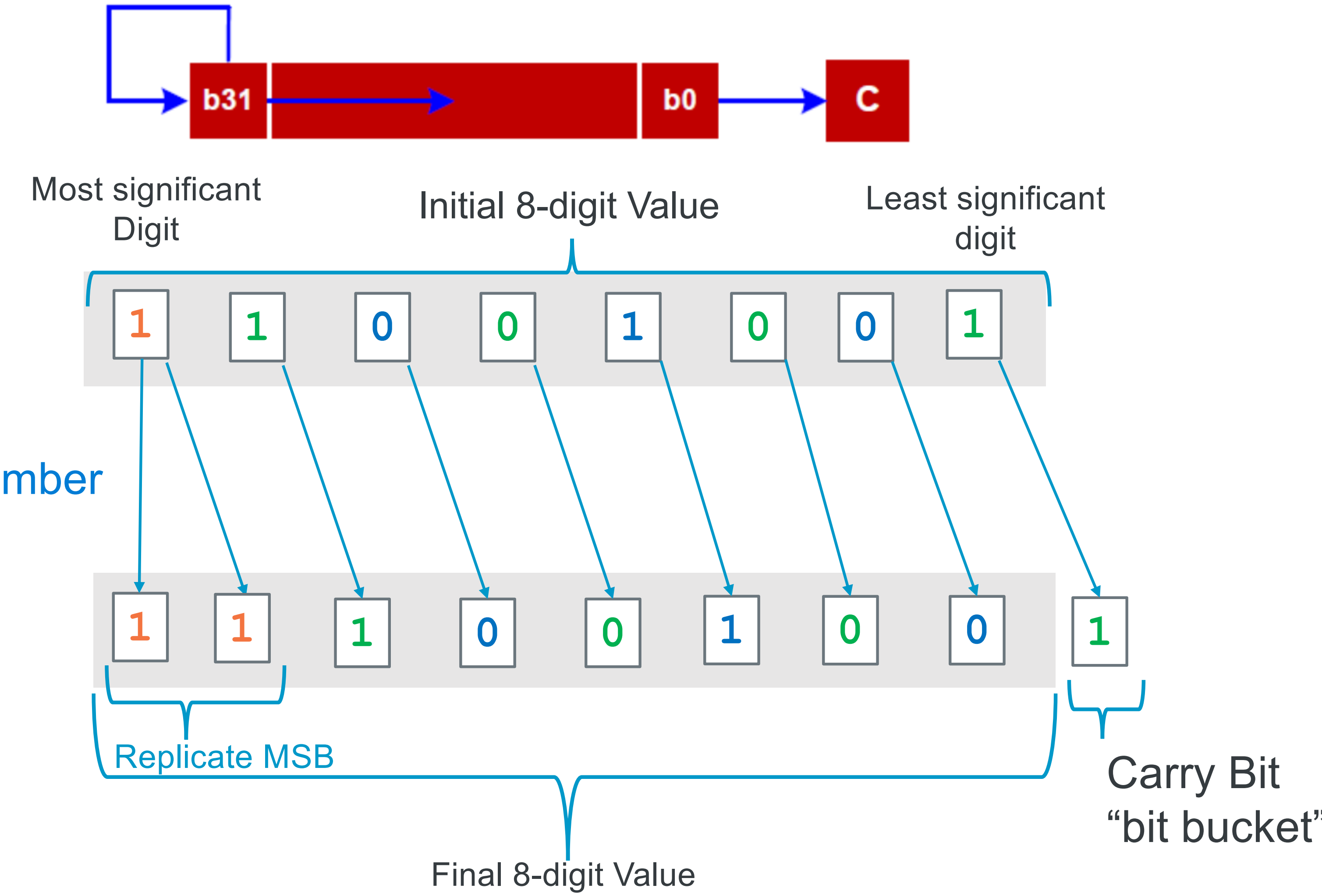
- n is number of bits to shift a variable x of width w bits
- Shifts by `n < 0` or `n ≥ w` are *undefined*
- Left shift (`x << N`)
  - Shift N bits left, Fill with 0s on right
- **In C:** behavior of **>>** is determined by compiler
  - gcc: it depends on data type of `x` (signed/unsigned)
- Right shift (`x >> N`)
  - Logical shift (for unsigned variables)
    - Shift N bits right, Fill with 0s on left
  - Arithmetic shift (for signed variables) – Sign Extension
    - Shift N bits right while **Replicating** the most significant bit on left
    - Maintains sign of `x`
- **In Java:** logical shift is **>>>** and arithmetic shift is >>



x

Arithmetic Shift Right 1 Digit = Divide by 2 for 2's Complement Values

Binary
Positive Number



X

Number ranges: limits.h

- The include file *<limits.h>* defines various symbolic names where the names represent the various limits on resources that the implementation imposes on applications;

C Data Type	signed min	signed max	unsigned max
`char`	`SCHAR_MIN`	`SCHAR_MAX`	`UCHAR_MAX`
`short int`	`SHRT_MIN`	`SHRT_MAX`	`USHRT_MAX`
`int`	`INT_MIN`	`INT_MAX`	`UINT_MAX`
`long int`	`LONG_MIN`	`LONG_MAX`	`ULONG_MAX`

```
#include <limits.h>
int i = INT_MAX;
…
      printf("Max int is: %d\n", i);
…
```

- The standard C integer types were intended to allow code to be portable among machines with different inherent data sizes (word sizes), so each type may have different range of numbers on different machines

x

Excess N Bias Encoding Method

- **Excess, Bias (or offset) encoding** maps negative numbers to an unsigned (positive) integer range by adding an offset number (called the bias) to encode positive and negative numbers
  - **Most negative number maps to zero, most positive number maps to all 1's**
- **For example: Say we have a number that is limited to** 3 bits (0 to 7 unsigned)

Actual	-3	-2	-1	0	1	2	3	4
Bias	+ 3	+ 3	+ 3	+ 3	+ 3	+ 3	+ 3	+3
Biased Encoded	0	1	2	3	4	5	6	7

x

Excess Bias Encoding (As used in floating point numbers)

- Given a number in E bits, to divide the range in about 1/2 the following is used:

  excess N bias = $(2^{E-1} - 1)$ *(this is just one of many bias formulas)*

- **With this excess N Bias approach**: actual numbers range from most negative to most positive is: **-(bias) to bias+1**

- **So, for a number that is limited to** 4 bits (0 to 15 unsigned)
  - Then excess N bias = $2^{4-1} - 1 = 2^3 - 1$ = a bias of +7

actual	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8
bias	+7	+7	+7	+7	+7	+7	+7	+7	+7	+7	+7	+7	+7	+7	+7	+7
bias encoded	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

x

Signed Magnitude Examples (Sign bit is always MSB)

$$\underbrace{0}_{\text{positive}}\ \underbrace{110}_{6}$$

$$\underbrace{1}_{\text{negative}}\ \underbrace{011}_{3}$$

Examples (4 bits):

1 000 = -0?	0 000 = 0?
1 001 = -1	0 001 = 1
1 010 = -2	0 010 = 2
1 011 = -3	0 011 = 3
1 100 = -4	0 100 = 4
1 101 = -5	0 101 = 5
1 110 = -6	0 110 = 6
1 111 = -7	0 111 = 7

$$\underbrace{0}_{\text{positive}}\ \underbrace{0000000}_{0}$$

$$\underbrace{1}_{\text{negative}}\ \underbrace{0001100}_{12}$$

Examples Using Hex notation (8 bits):

0x00 = 0b00000000 is positive, because the sign bit is 0

0x7F = 0b01111111 is positive ($+127_{10}$)

0x85 = 0b10000101 is negative (-5_{10})

0x80 = 0b10000000 is negative… also zero

Another Way to Look at 2's Complement Encoding

- A 2's compliment value can be thought of as using a slightly different **bias encoding** for negative numbers only (more negative values): -2^{w-1}
- The leftmost bit is then interpreted as a decision to apply the bias (if 1) or not (if 0)
  - 1 apply the bias
  - 0 do not apply the bias
- For example, for a 4-bit number (w = 4) , the negative number bias weight would be $= -2^{4-1} = -2^3 = -8$

2's	1000	1001	1010	1011	1100	1101	1110	1111	0000	0001	0010	0011	0100	0101	0110	0111
3 bit	000	001	010	011	100	101	110	111	000	001	010	011	100	101	110	111
decimal	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
+Bias	-8	-8	-8	-8	-8	-8	-8	-8	0	0	0	0	0	0	0	0
Actual	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7

Observe: adding +1 makes the number more positive for both negative and positive numbers

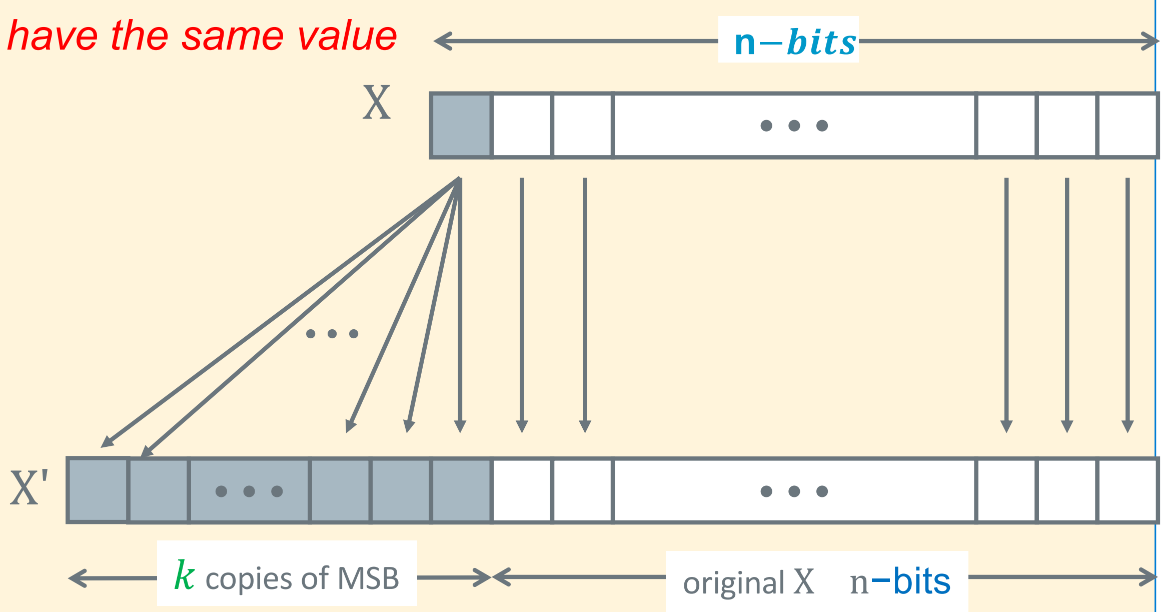
x

Sign Extension (how type promotion works)

- Sometimes you need to work with integers encoded with different number of bits

  8 bits (char) -> (16 bits) **`short`** -> (32 bits) **`int`**

- **Sign extension increases the number of bits: n-bit** wide signed integer X, ***EXPANDS*** to a ***wider*** $\mathrm{n}-$bit + k-bit signed integer X' where *both have the same value*



Unsigned

- Just add leading zeroes to the left side

Two's Complement Signed:

- If positive, add leading zeroes on the left
  - Observe: Positive stay positive
- If negative, add leading ones on the left
  - Observe: Negative stays negative

x

Example: Two's Complement Sign or bit Extension - 1

- Adding 0's in front of a positive numbers does not change its value

```
     7    =    0111
extend to
8 bits
           00000111
Number is still 7
```

```
     1    =    0001
extend to
8 bits
           00000001
Number is still 1
```


x

Example: Two's Complement Sign or bit Extension -2

- Adding 1's if front of a negative number does not change its value

```
     7 = 0111
invert = 1000
add 1  +    1
    -7   1001
```

```
   -7     =     1001
extend to
8 bits
              11111001
```

```
     7 = 00000111
invert = 11111000
add 1  +        1
    -7   11111001
```


x

Example: Two’s Complement Sign or bit Extension - 3

- Adding 1's if front of a negative number does not change its value

```
        1 = 0001

   invert = 1110
   add 1  +    1
       -1   1111
```

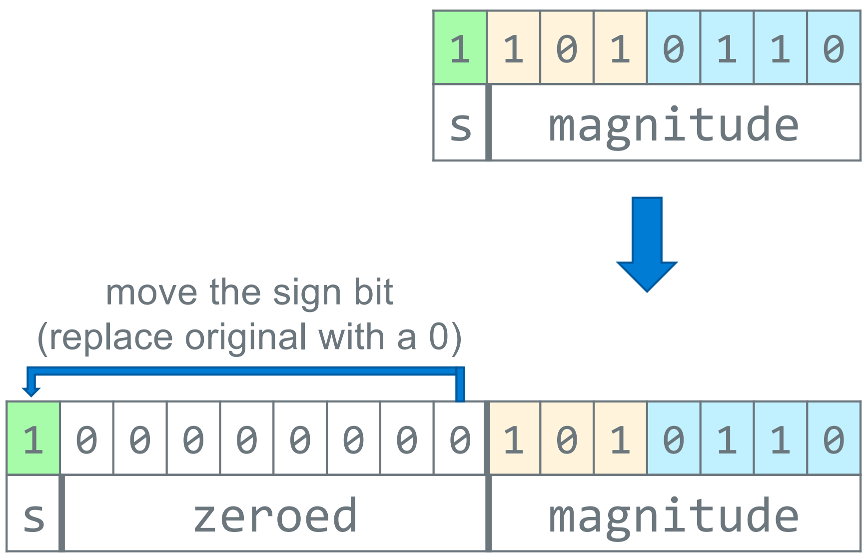
```
    -1     =         11111111
extend to
16 bits
             1111111111111111
```

```
    -1     =     1111
extend to
8 bits
             11111111
```


x

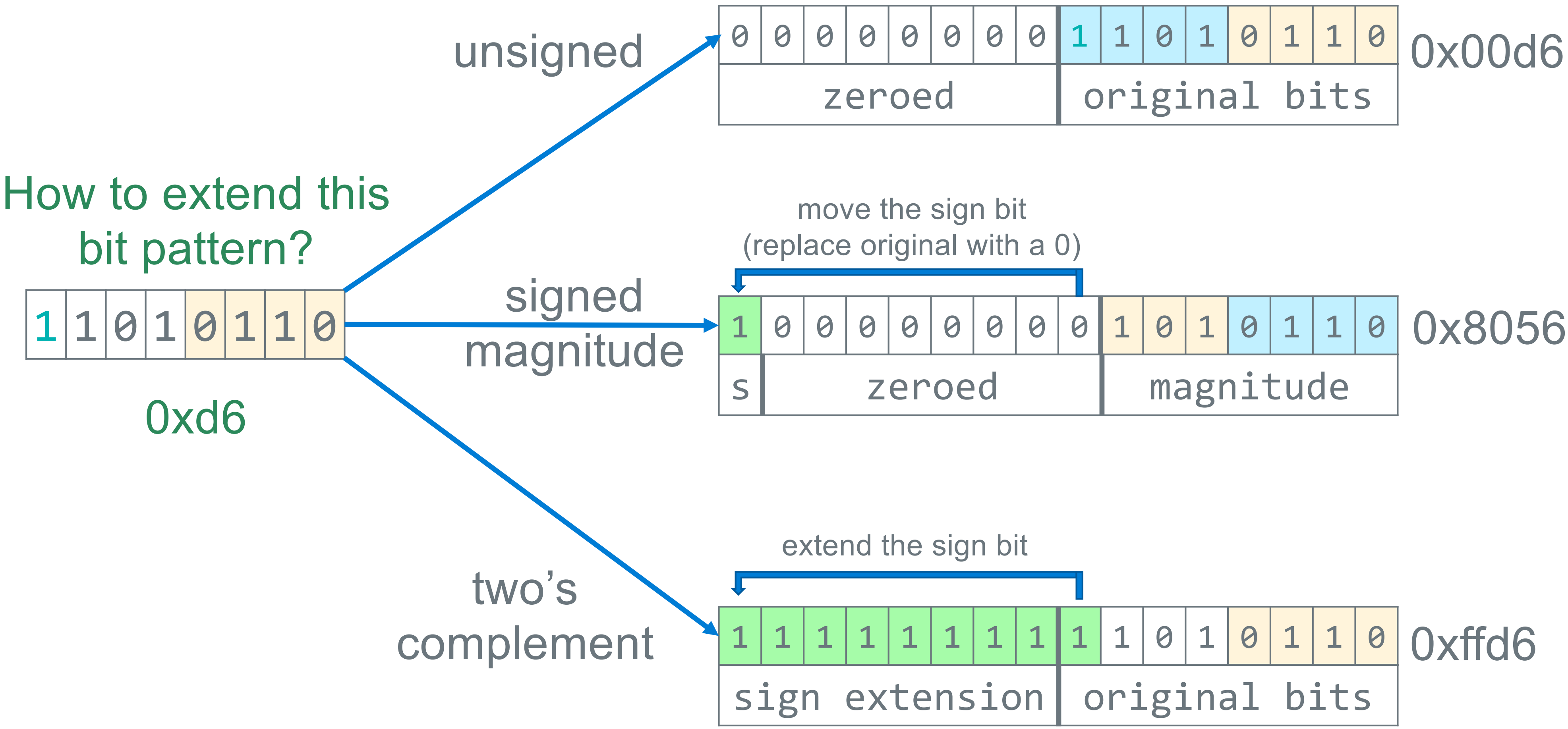
Sign Extension Signed Magnitude number

- Just move the sig bit and expand the magnitude with zeros to the left



x

Interpreting and extending with Different representations



Scientific Notation Decimal

$$+\overbrace{6.02}^{\text{mantissa}}{}_{10} \times 10^{23}$$

sign

decimal point

exponent

radix (base)

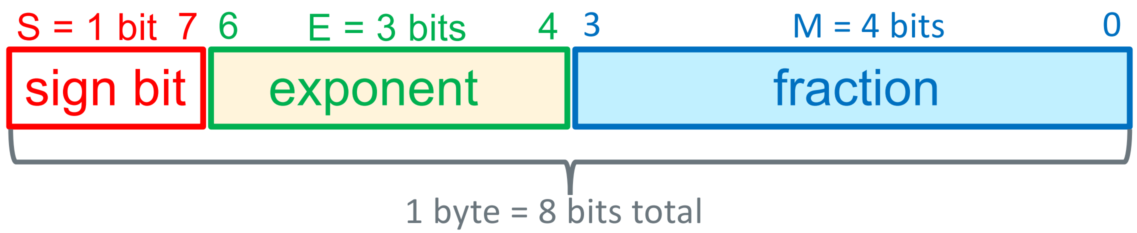
- *Scientific Normalized form*:

  exactly one digit (non-zero) to left of decimal point

- Alternatives to representing 1/1,000,000,000
  - Normalized: 1.0×10^{-9}
  - Not normalized: 0.1×10^{-8}, 10.0×10^{-10}

x

Floating Point Number in a Byte (Not A Real Format)



- **Mantissa encoding: = 1.[xxxx] encoded as an unsigned value**
- **Exponent encoding:** 3 bits encoded as an unsigned value using bias encoding
  - **Bias encoding** = **$(2^{E-1} - 1)$**
  - 3 bits for the bias we have $2^{3-1} - 1 = 2^2 - 1$ = a bias of 3
  - **With a Bias of 3**: positive and negative numbers range: small to large is: 2^{-3} to 2^4

Actual	-3	-2	-1	0	1	2	3	4
Bias	+ 3	+ 3	+ 3	+ 3	+ 3	+ 3	+ 3	+3
Biased	0	1	2	3	4	5	6	7

x

Floating Point Number (8-bits) Number Range: 2^{-3} to 2^{4}

S = 1 bit	E = 3 bits	M = 4 bits
sign bit	exponent	fraction

S = 1 bit	E = 3 bits	M = 4 bits
0	000	0000

0.0 <u>Special case</u> in this simple model we <u>do not</u> put back the "hidden bit"

S = 1 bit	E = 3 bits	M = 4 bits
0	000	0001

Smallest Non-zero Positive
0.0010001 = 1/8 + 1/128 = 0.1328125 base 10

S = 1 bit	E = 3 bits	M = 4 bits
0/1	111	1111

Largest Positive/Negative
1.1111 x 2^4 = 11111 = 31 base 10

S = 1 bit	E = 3 bits	M = 4 bits
1	000	0000

Smallest (closest to zero) Number
1.0000 x 2^{-3} = 0.001000 = 1/8 = -0.125 base 10

Note: Orange is hidden bit added back

x

Decimal to Float

Bias of 3

7	6 … 4	3 … 0
s	exponent (3 bits)	fraction (4 bits)

Step 1: convert from base 10 to binary (absolute value)

$$-0.375\,(\text{decimal}) = 0000.0110_2$$

Step 2: Find out how many places to shift to get the number into the normalized 1.xxxx mantissa format

$$0000.0110_2 = 1.1000 \times (2^{-2})_{\text{base }10}$$

exponent: -2_{10} + bias of 3_{10} = 1_{10} = 0b001 for the exponent (after adding the bias)

Step 3: Use as many digits that fit to the right of the decimal point in the fractional .xxxx part

1.1000

Step 4: Sign bit

positive sign bit is 0

negative sign bit is 1

s	exponent	fraction
1	0b001	0b1000
0x9		0x8

= 0x98

x

Float to Decimal

7	6 Bias of 3 4	3 0
s	exponent (3 bits)	fraction (4 bits)

Step 1: Break into binary fields

`0x45 =`

0x4		0x5
s	exponent	fraction
0	0b100	0b0101

Step 2: Extract the unbiased exponent

`0b100` $= 4_{\text{base }10} - \text{bias of } 3_{10} = 1_{10}$ for the exponent (bias removed)

Step 3: Express the mantissa (restore the hidden bit)

`1.0101`

Step 4: Apply the **unbiased** exponent

$1.0101_{\text{base }2} \times (2^1)_{\text{base }10} = 10.101$

Step 5: Convert to decimal

$10.101 = 2.625_{\text{base }10}$

Step 6: Apply the Sign

$+\ 2.625_{\text{base }10}$

x

IEEE “754” Floating Point Double and Single Precision

Single Precision (C float)

31	30 … 23	22 … 0
sign	Exponent (8 bits) Uses a Bias of 127	fraction (23 bits)

$$Bias\ is\ (2^{8-1}-)\ = 127$$

$$single\ precision\ floating\ point\ number = (-1)^{s}\ x\ 2^{E-127}$$ x 1.fraction

Double Precision (C Double)

63	62 … 52	51 … 0
sign	Exponent (11 bits) Uses a Bias of 1023	fraction (52 bits)

$$bias\ is\ (2^{11-1}-1)\ = 1023$$

$$double\ precision\ floating\ point\ number = (-1)^{s}\ x\ 2^{E-1023}$$ x 1.fraction

Decimal to IEEE Single Precision Float

31	30 … 23	22 … 0
sign	Exponent (8 bits) Bias is 127	fraction (23 bits)

Step 1: convert from base 10 to binary (absolute value)

`-13.375(decimal) =  1101.0110`

Step 2: Find out how many places to shift to get the number into the normalized 1.xxxx mantissa format

`1101.0110 = 1.1010110 x (2`^{3}`)` base 10

`3 + bias of 127 = 130 for the exponent = 0b1000 0010`

Step 3: Use as many digits that fit to the right of the decimal point in the fractional .xxxx part (0 pad)

`1.1010110 0000 0000 0000 0000`

Step 4: If the sign is positive sign bit is 0, otherwise it is 1

s	exponent	fraction
1	100 0001 0	101 0110 0000 0000 0000 0000

$\underbrace{1\,100}_{0xc}$ $\underbrace{0001}_{0x1}$ $\underbrace{0\,101}_{0x5}$ 0x6 0x0 0x0 0x0 0x0 `= 0xc1560000`

x

IEEE Single Precision Float to Decimal

31	30 … 23	22 … 0
sign	Exponent (8 bits) Bias is 127_{10}	fraction (23 bits)

Step 1: Break into binary fields and expand as needed

`0xc0b00000 =`

0xc		0x0	0xb		0x0	0x0	0x0	0x0	0x0
1	100	0000	1	011	0000	0000	0000	0000	0000

Step 2: Find the exponent

`0b10000001` $= 129_{\text{base } 10}$ – bias of $127_{10} = 2_{10}$ exponent with bias added

Step 3: Express the mantissa (restore the hidden bit)

`1.0110`

Step 4: Apply the exponent

$1.0110 \times (2^2)_{\text{base } 10} = 101.10$

Step 5: Convert to decimal

$101.10 = 5.5$

Step 6: Apply the Sign

-5.5

x

Reference: 8-Bit Overflow Examples

Unsigned Integer

cout									
1	1	1	1	1	1	1	0		carries
	1	1	1	0	1	0	1	0	$=234_{10}$
+	0	0	1	1	0	1	1	0	$=54_{10}$
	0	0	1	0	0	0	0	0	$=32_{10}$

Because carry-out bit is 1 (and dropped), overflow is detected

Two's Complement

cout	cin								
0	1	1	1	1	1	1	0		carries
	0	1	1	0	1	0	1	0	$=106_{10}$
+	0	0	1	1	0	1	1	0	$=54_{10}$
	1	0	1	0	0	0	0	0	$=-96_{10}$

Both operands are positive, but resulting sign is negative
see that cout != cin at the MSB
overflow is detected

Two's Complement

cout	cin								
1	1	1	1	1	1	1	0		carries
	1	1	1	0	1	0	1	0	$=-22_{10}$
+	0	0	1	1	0	1	1	0	$=54_{10}$
	0	0	1	0	0	0	0	0	$=32_{10}$

Unlike unsigned arithmetic, no overflow even though the carry-out bit is 1.
As the operand's signs differ, overflow is not possible (cout == cin)

X

Decimal to Float

7	6 Bias of 3 4	3 0
s	exponent (3 bits)	fraction (4 bits)

Step 1: convert from base 10 to binary (absolute value)

`6.625(decimal) =  0110.1010`

Step 2: Find out how many places to shift to get the number into the normalized 1.xxxx mantissa format

`0110.1010` normalizes to -> $1.101010 \times (2^2)_{\text{base } 10}$

exponent: 2_{10} + a bias of 3_{10} = 5_{10} = 0b101 for the exponent (after adding the bias)

Step 3: Use as many digits to the right of the decimal point that will fit in the fractional .xxxx part

1.101010 (we will truncate drop the trailing 10, Real FP use complex rounding approaches)

Step 4: Sign bit
positive sign bit is 0
negative sign bit is 1

s	exponent	fraction
0	0b101	0b1010
0x5		0xa

= 0x5a

x