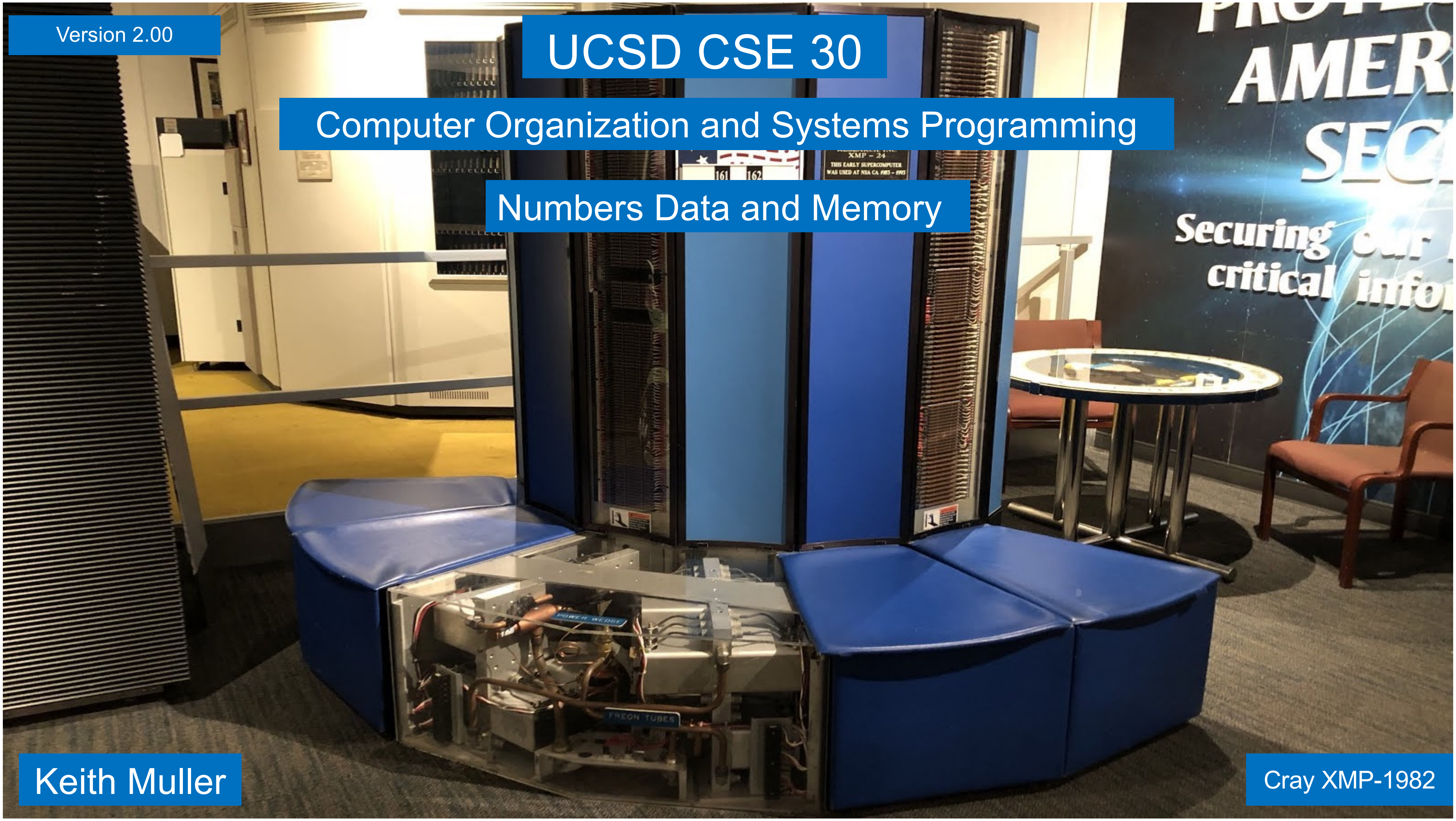
Version 2.00

UCSD CSE 30

Computer Organization and Systems Programming

Numbers Data and Memory



Keith Muller

Cray XMP-1982

Positive Number (unsigned) in 4 bits

- Real hardware has a fixed number of bits to store numbers (pi-cluster is 32 bits)
- There are only 2^n distinct values in n bits
- This limits the range of positive number to be 0 (unsigned min) to 2^n - 1 (unsigned max)

Hex digit	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0b0000	0b0001	0b0010	0b0011	0b0100	0b0101	0b0110	0b0111

umin

Hex digit	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
Decimal value	8	9	10	11	12	13	14	15
Binary value	0b1000	0b1001	0b1010	0b1011	0b1100	0b1101	0b1110	0b1111

umax

x

Unsigned Integers (positive numbers) with Fixed # of Bits

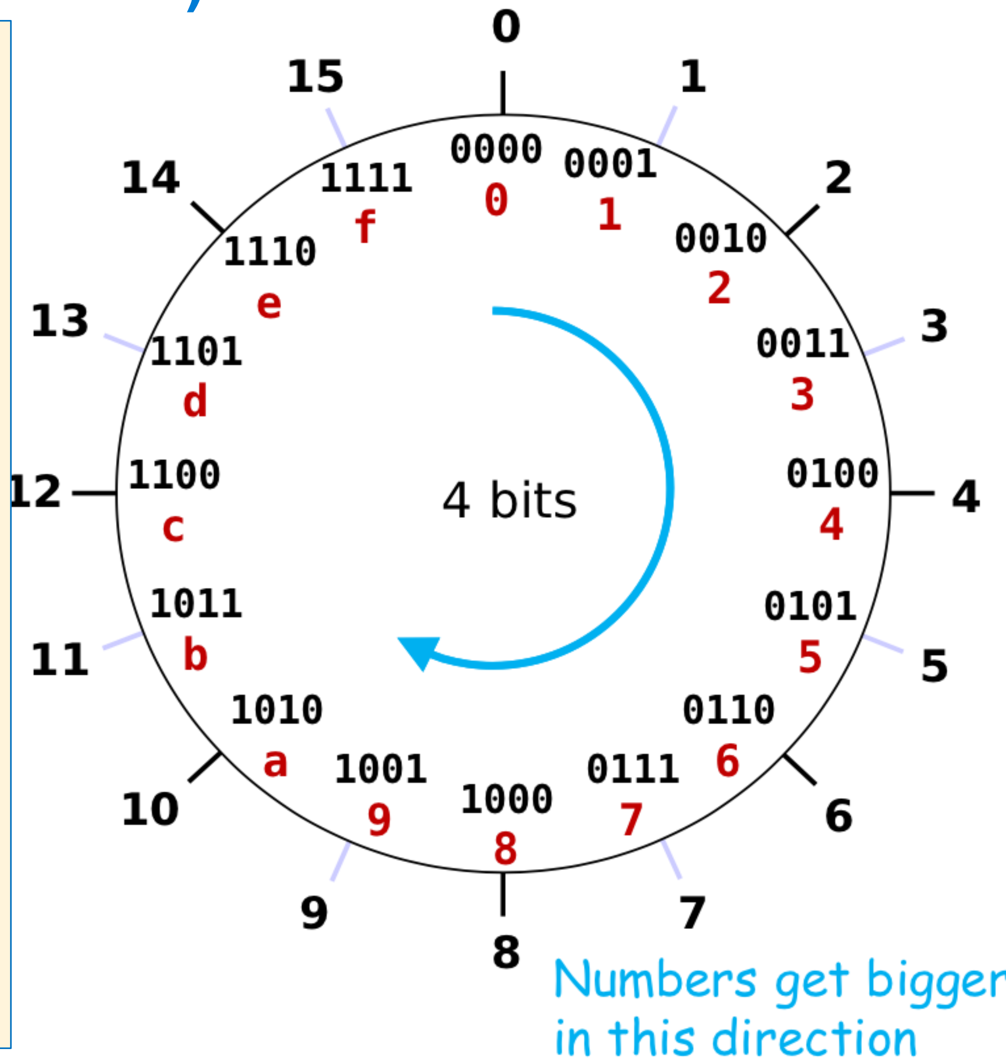
- 4 bits is 2^4 = ONLY 16 distinct values
- **Modular** (C operator: %) or clock math
  - Numbers start at 0 and “wrap around” after 15 and go back to 0
- Keep adding 1

  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



x

Unsigned Binary Number: Addition in 4 bits

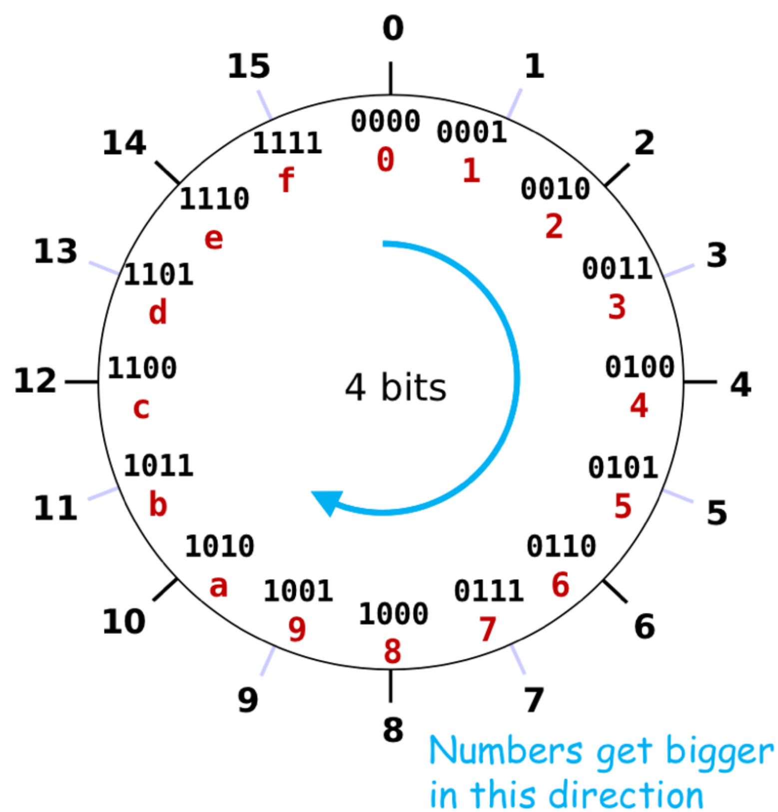
Be Aware in Binary

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

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

Carry Bit

carries	0	1	1	1		
		0	0	0	1	1
+		0	1	1	1	7
sum		1	0	0	0	= 8



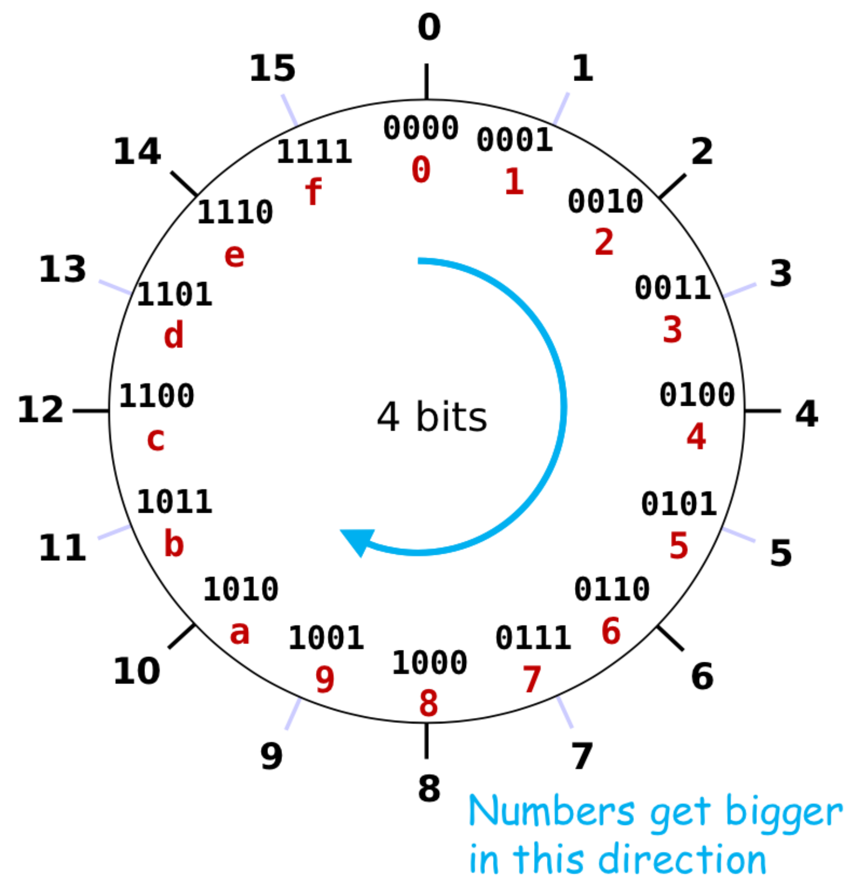
x

Unsigned Binary Number: Subtraction in 4 bits

Be Aware in Binary	
1 - 1 = 0	base 10: (1 - 1 = 0)
10 - 1 = 1	base10: (2 – 1 = 1)

Borrows

$$
\begin{array}{rcccc}
 & 0 & \cancel{1} & {}^{1}0 & 1 \\
- & 0 & 0 & 1 & 1 \\
\hline
\text{sum} & 0 & 0 & 1 & 0
\end{array}
\quad = \quad
\begin{array}{r}
5 \\
-\ 3 \\
\hline
2
\end{array}
$$



x

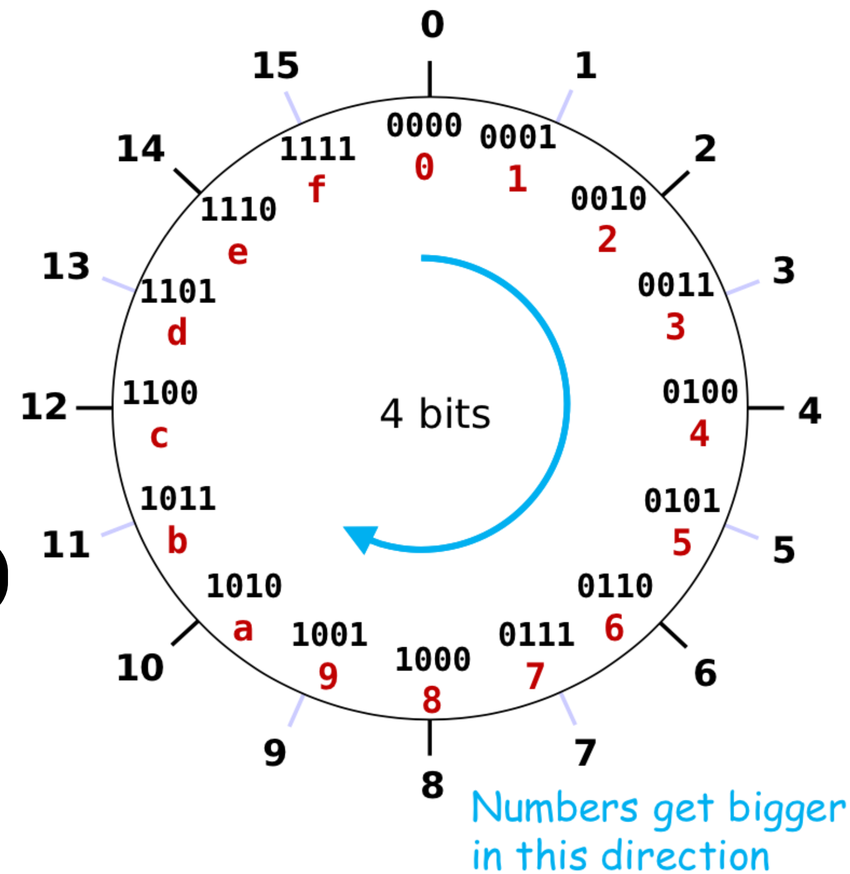
Unsigned Binary Number: Addition in 4 bits – Overflow!

Be Aware in Binary

1 + 1 = 10	base 10: (1 + 1 = 2)
1 + 1 + 1 = 11	base10: (1 + 1 + 1 = 3)

Carry Bit

carries	1	1	1			
		1	0	1	0	10
+		0	1	1	1	7
sum		0	0	0	1	≠ 17



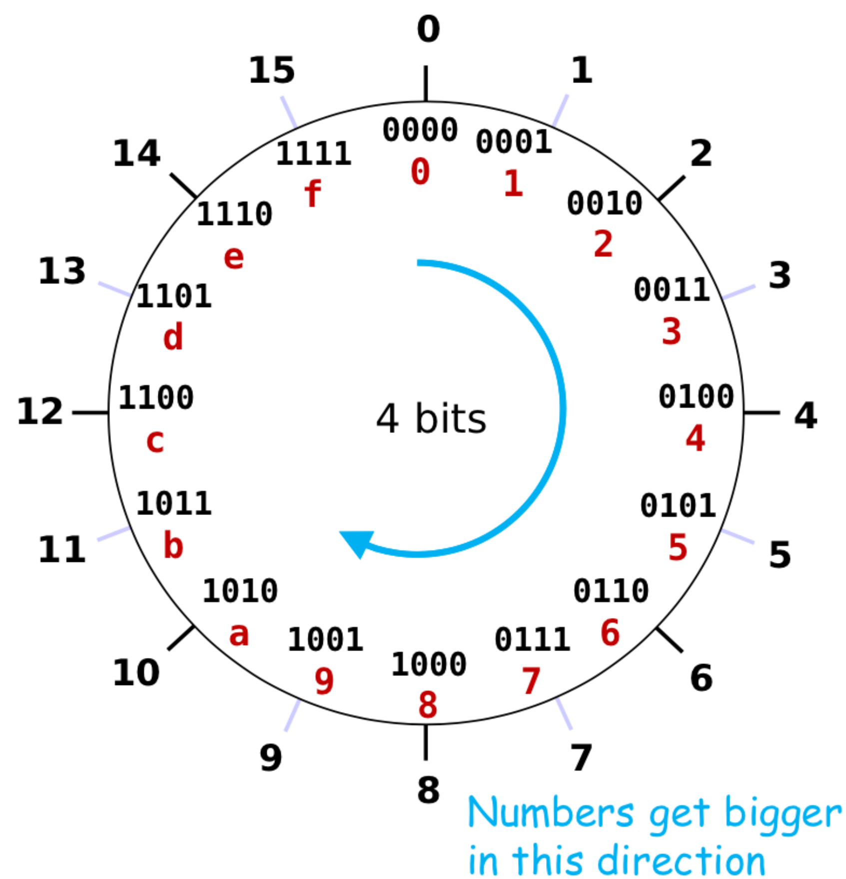
x

Unsigned Binary Number: Subtraction in 4 bits – Overflow!

Be Aware in Binary	
1 - 1 = 0	base 10: (1 - 1 = 0)
10 - 1 = 1	base10: (2 – 1 = 1)

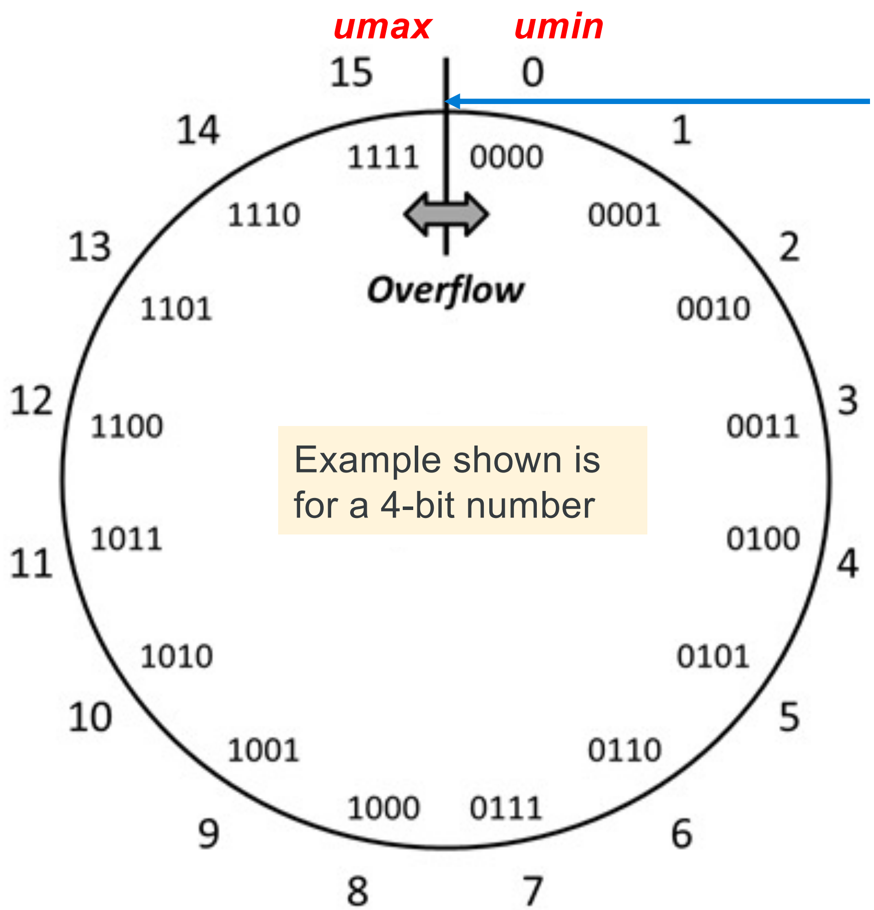
Borrows

$$
\begin{array}{rcccccc}
 & \cancel{1}\,1 & 1\,0 & & 1 & & \\
 & \cancel{0} & \cancel{1} & 0 & 1 & & 5 \\
- & 0 & 1 & 1 & 1 & & 7 \\
\hline
\text{sum} & 1 & 1 & 1 & 0 & \neq & -2
\end{array}
$$



x

Overflow: Going Past the Boundary Between umax and umin



Overflow with unsigned numbers:
Occurs when an arithmetic result (from addition or subtraction for example) 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

Problem: How to Encode Both Positive and Negative Integers

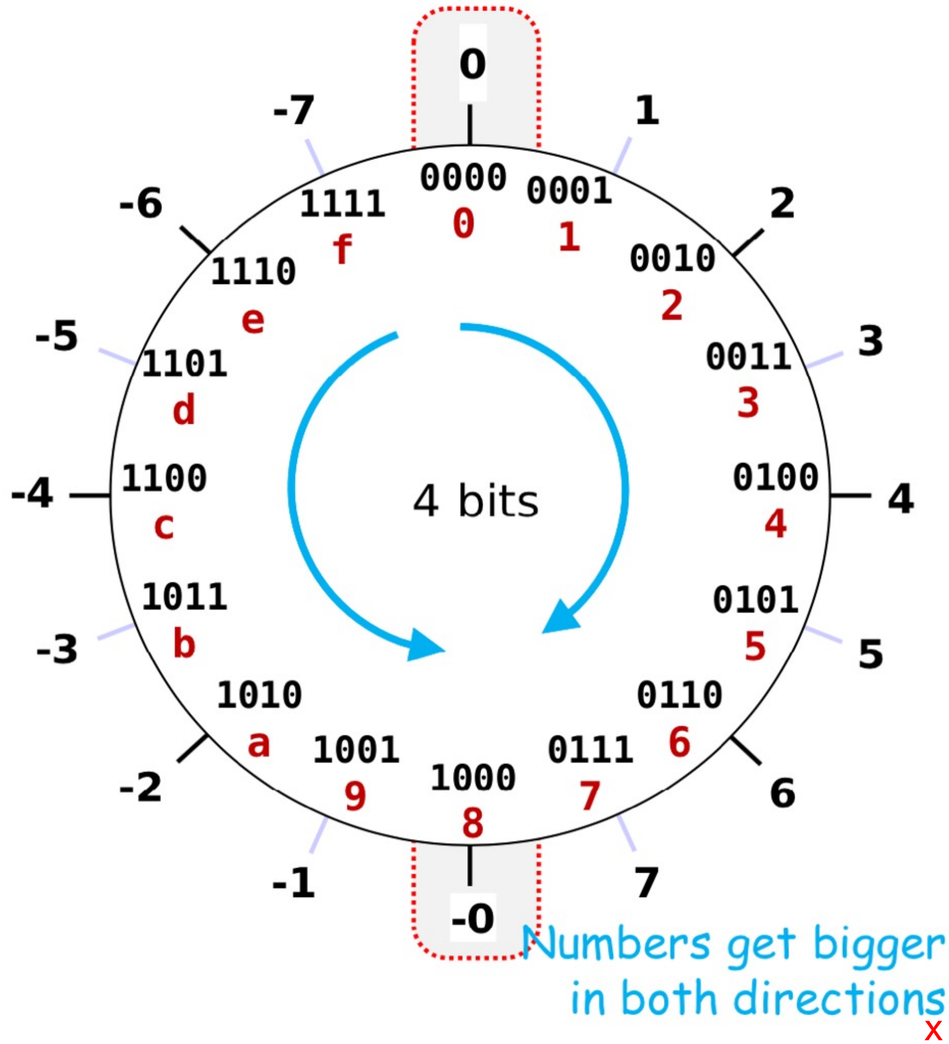
- How do we represent the negative numbers within a fixed number of bits?
  - Allocate some bit patterns to negative and others to positive numbers (and zero)
- $\mathbf{2^n}$ distinct bit patterns to encode positive and negative values
- **Unsigned values:** $0 \ldots 2^{n}–1$ ← -1 comes from counting 0 as a "positive" number
- **Signed values:** $-2^{n-1} \ldots 2^{n-1}–1$ (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)
  - Positive and Negatives "*increment*" (+1) in the opposite directions!



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

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

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

x

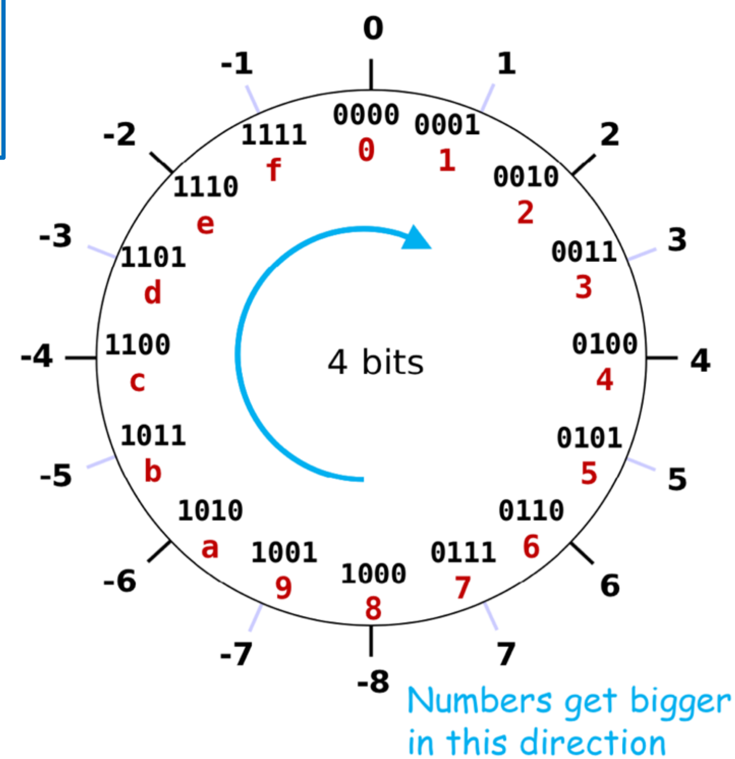
Two's Complement: The MSB Has a *Negative Weight*

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

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



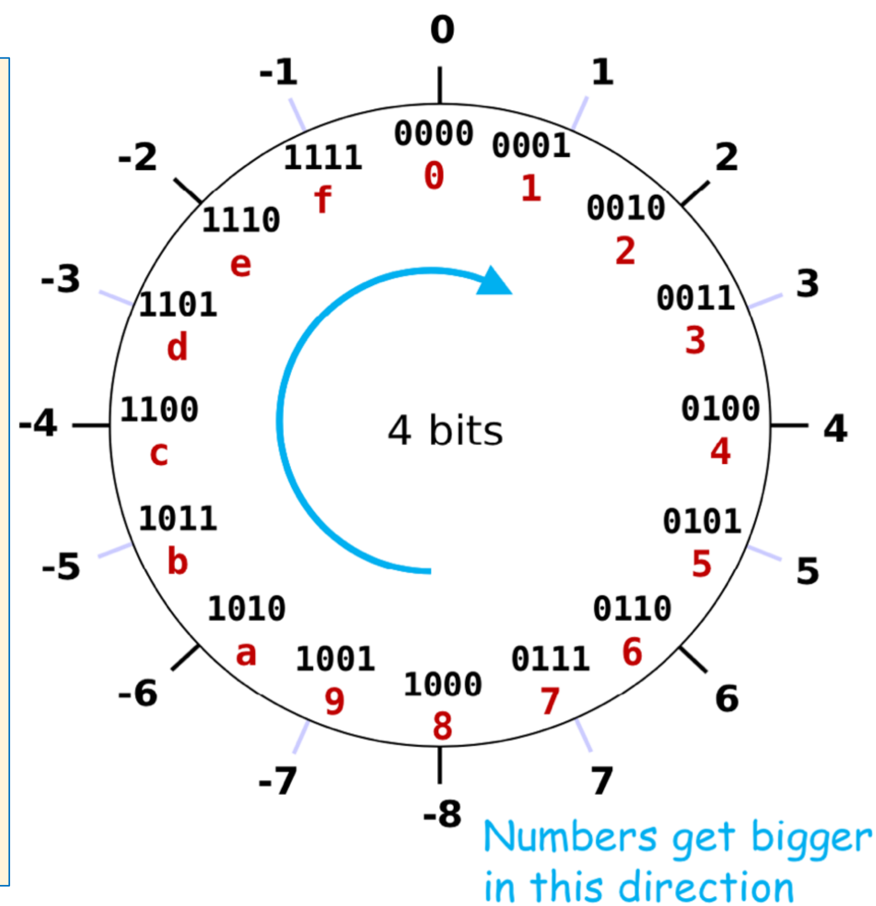
x

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 1 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

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

  = $\mathbf{2^n - 1}$

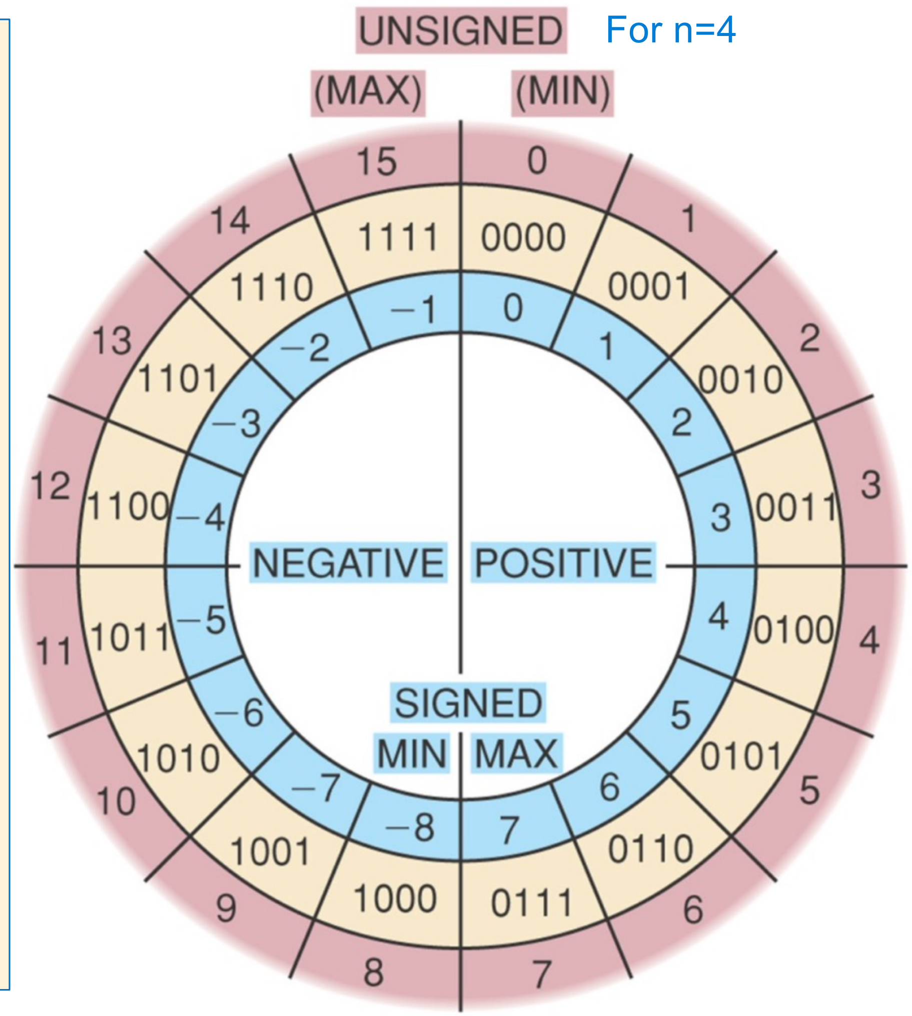
- **Two's Complement Range**

  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
```

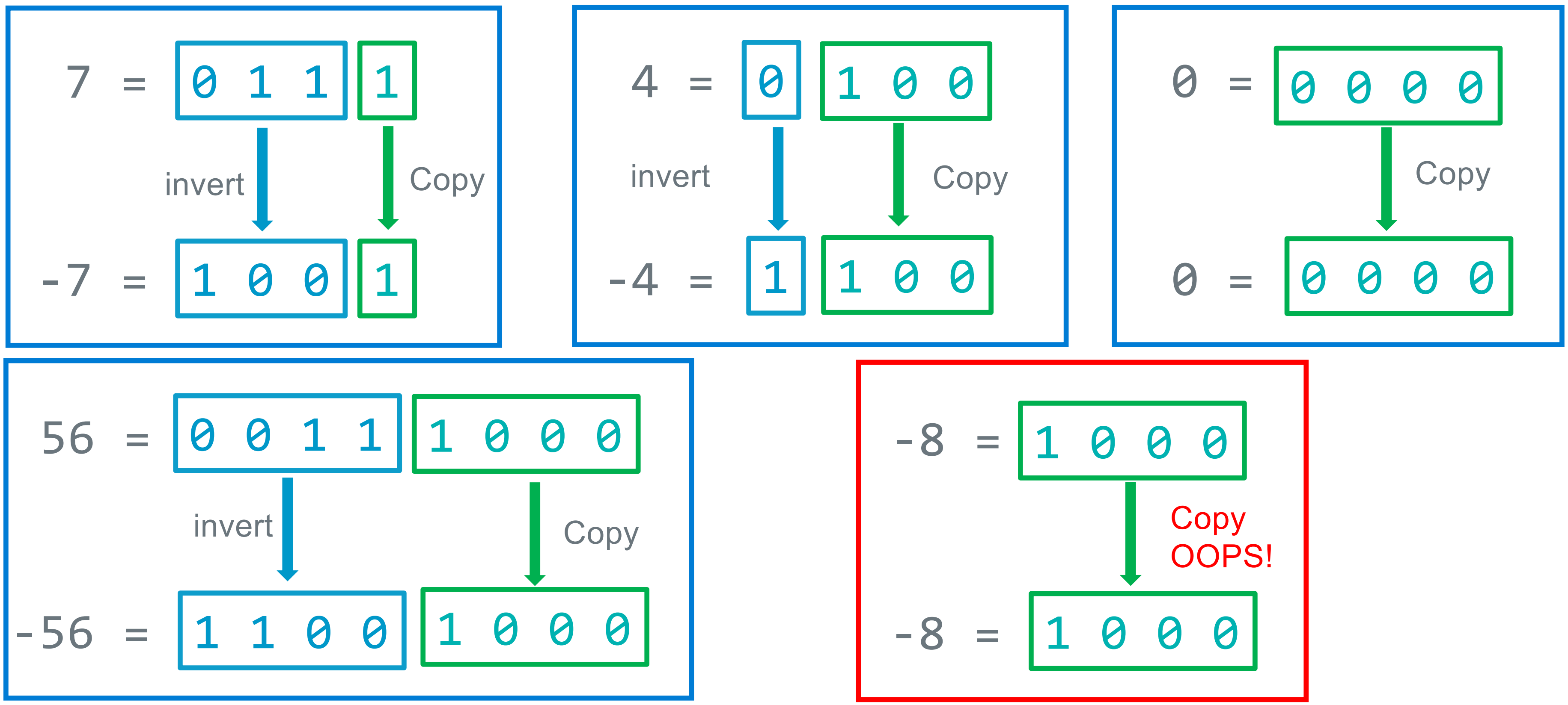
```
     -1 = 1111
invert  = 0000
add 1   +    1
      1   0001
```

```
     -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 $\overset{b_7}{0}\ \overset{b_6}{1}\ \overset{b_5}{1}\ \overset{b_4}{0}\ \overset{b_3}{0}\ \overset{b_2}{1}\ \overset{b_1}{0}\ \overset{b_0}{1}_{(\text{base } 2)}$ 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
0	+ 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

b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0

What is 1 1 1 0 0 1 0 1 (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
0	+ 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 + 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
```

2’s complement first and then add

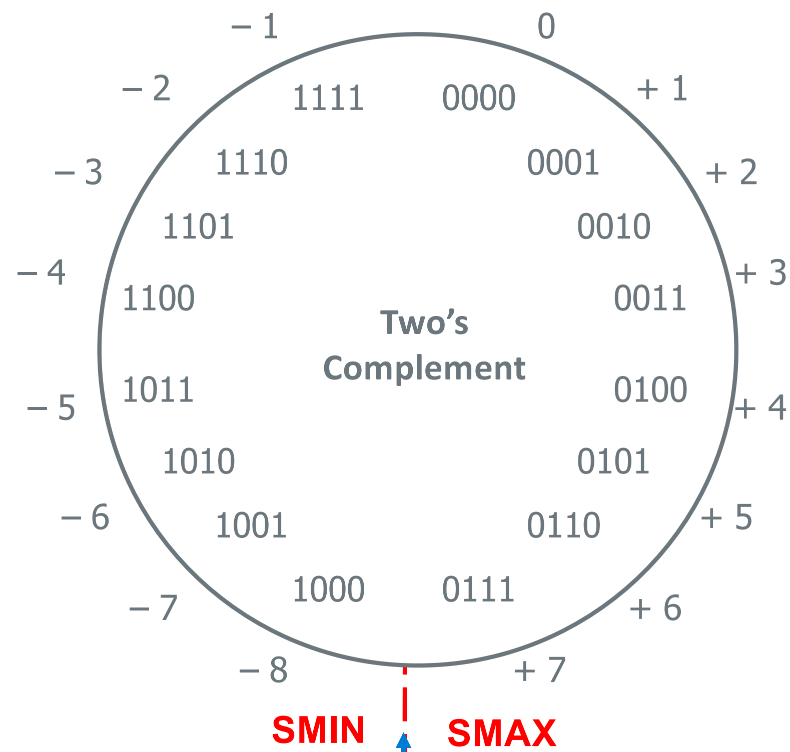
```
               x = 0 1 0 1 0 0 1 1
          + (-y) = 1 1 1 1 0 1 0 1
                   ---------------
x - y = x +(-y)  = 0 1 0 0 1 0 0 0
```


x

Two’s Complement Positive Overflow

- **4-bit** Two’s complement numbers (positive overflow)

```
Cout
 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

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

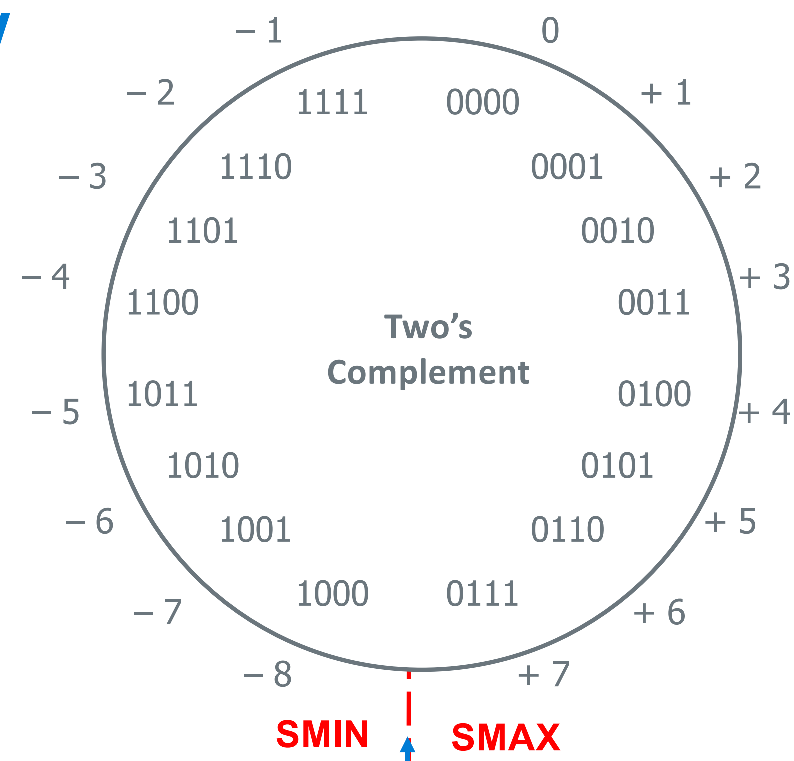
x

Two’s Complement Negative Overflow

- **4-bit** Two’s complement numbers (negative overflow)

carry bit is dropped from result

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



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

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

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

Sign Extension in C: Type casts

- Convert from smaller to larger integral data types
- C and Java automatically performs sign extension
- Example (on pi-cluster with 32-bit int and 16-bit short)

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
    signed char c = -1;
    signed int i = c;
    unsigned char d = 1;
    unsigned int j = d;
    printf("c decimal = %hd\n", c);
    printf("c = 0x%hhx\n", c);
    printf("i decimal = %d\n", i);
    printf("i = 0x%x\n", i);
    printf("\nd decimal = %hd\n", d);
    printf("d = 0x%hhx\n", d);
    printf("j decimal = %d\n", j);
    printf("j = 0x%x\n", j);
    return EXIT_SUCCESS;
}
```

```
%./a.out
c decimal = -1
c = 0xff
i decimal = -1
i = 0xffffffff

d decimal = 1
d = 0x1
j decimal = 1
j = 0x1
```

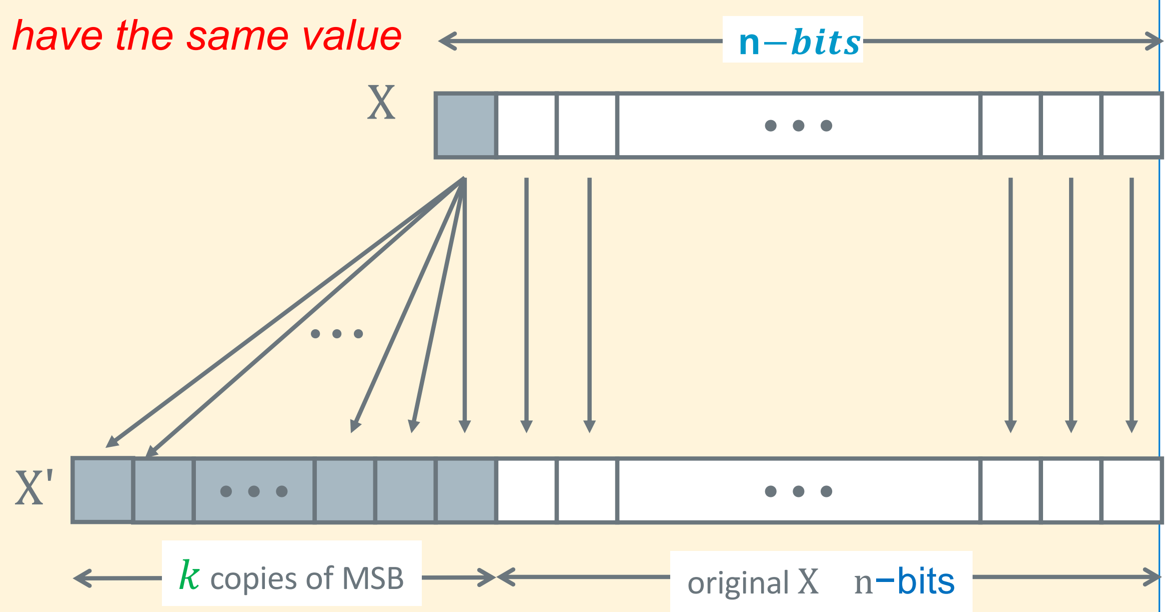

x

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*** 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
```

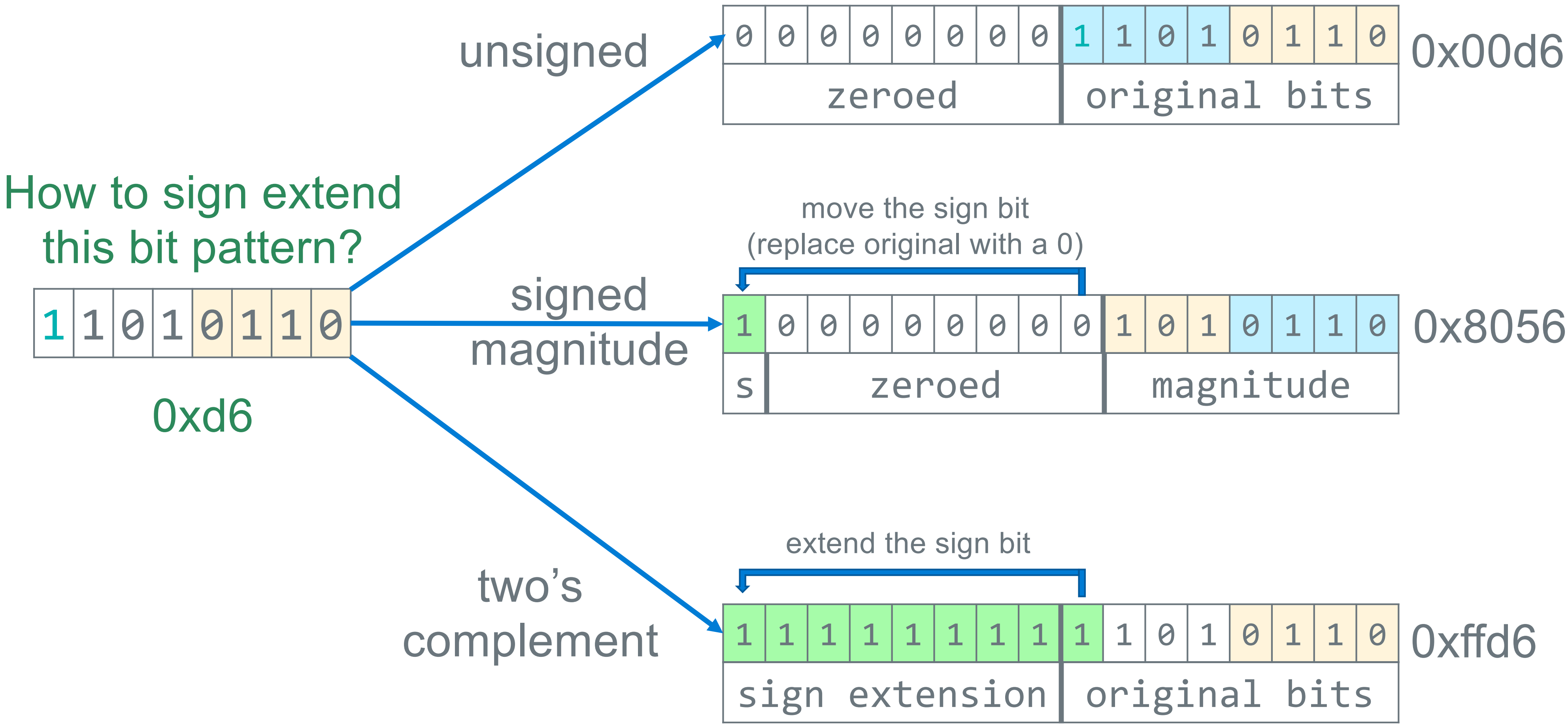
```
   -7     =     1001
extend to
8 bits
            11111001
```

```
1001 = -8 + 1 = -7
11111001 =
(-128 + 64 + 32 + 16 + 8) + 1
= -8 + 1 = -7
```

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


x

Sign Extension Under Different representations



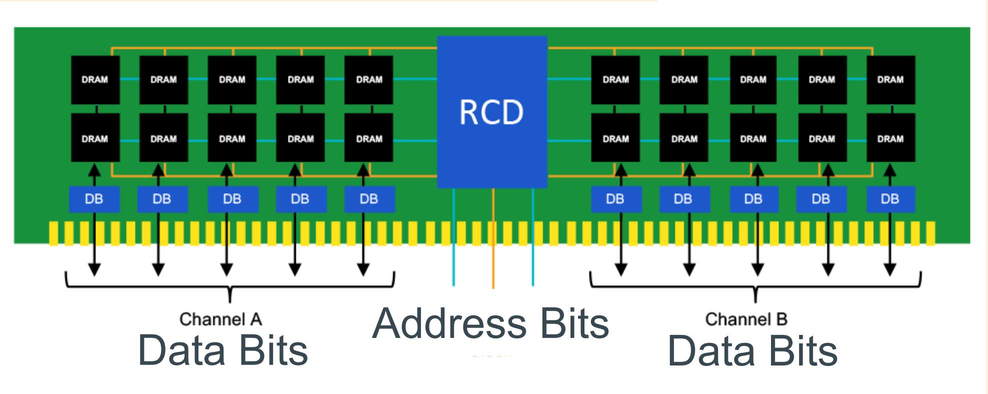
x

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 = $\mathbf{2^{32}}$ = 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

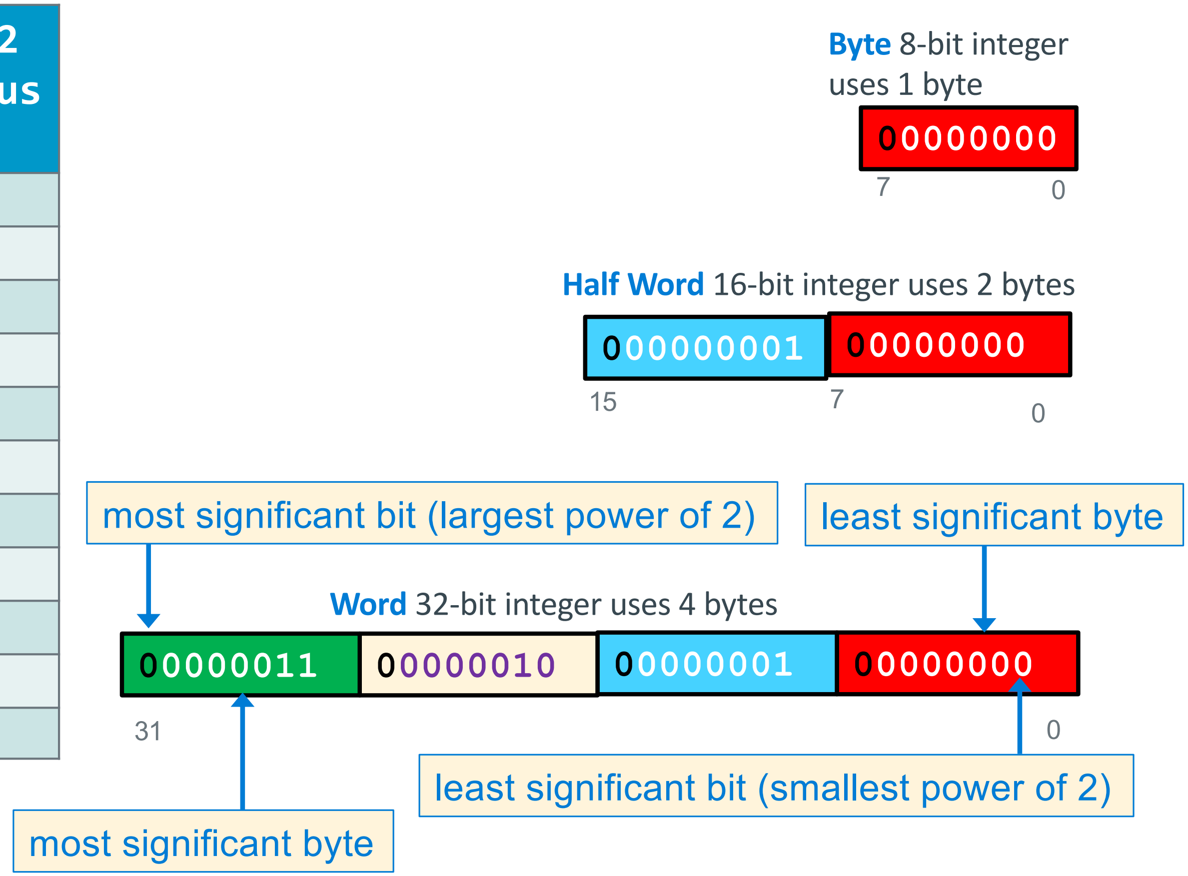
Memory dimm



x

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

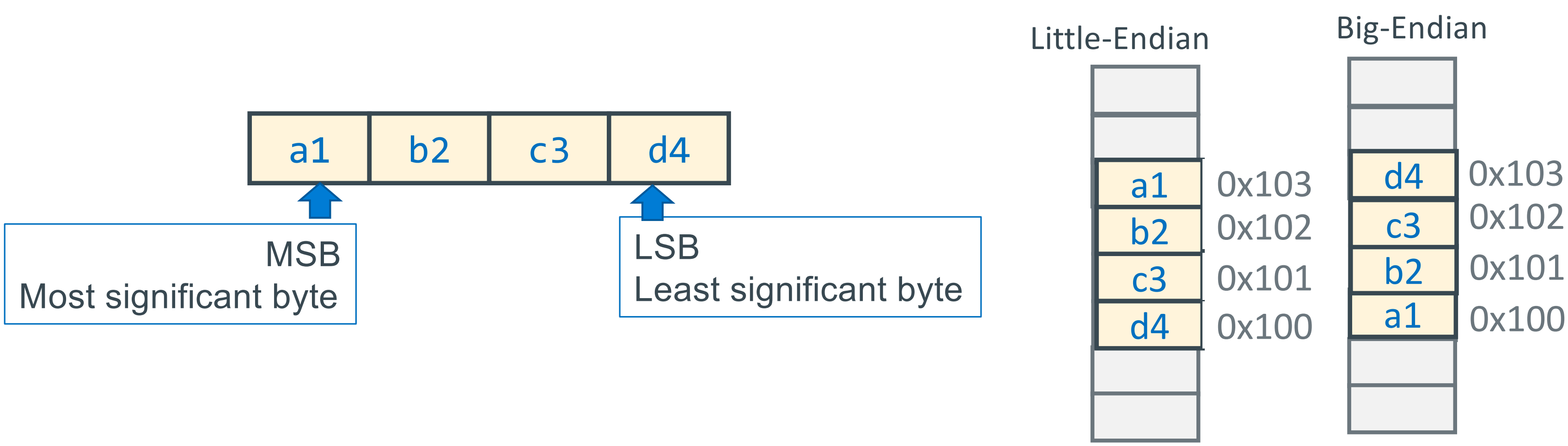
C Data Type	AArch-32 contiguous Bytes
char (arm unsigned)	1
short int	2
unsigned short int	2
int	4
unsigned int	4
long int	4
long long int	8
float	4
double	8
long double	8
pointer *	4



x

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

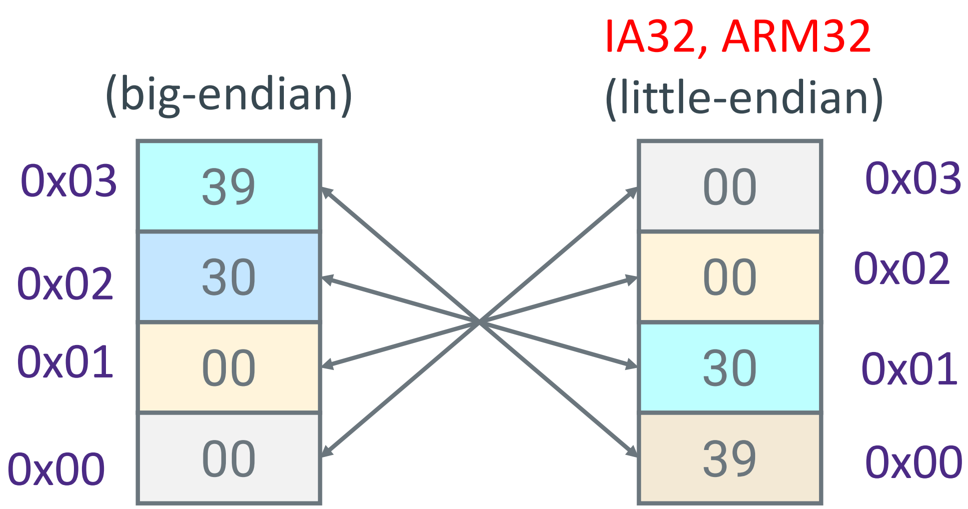


x

Byte Ordering Example

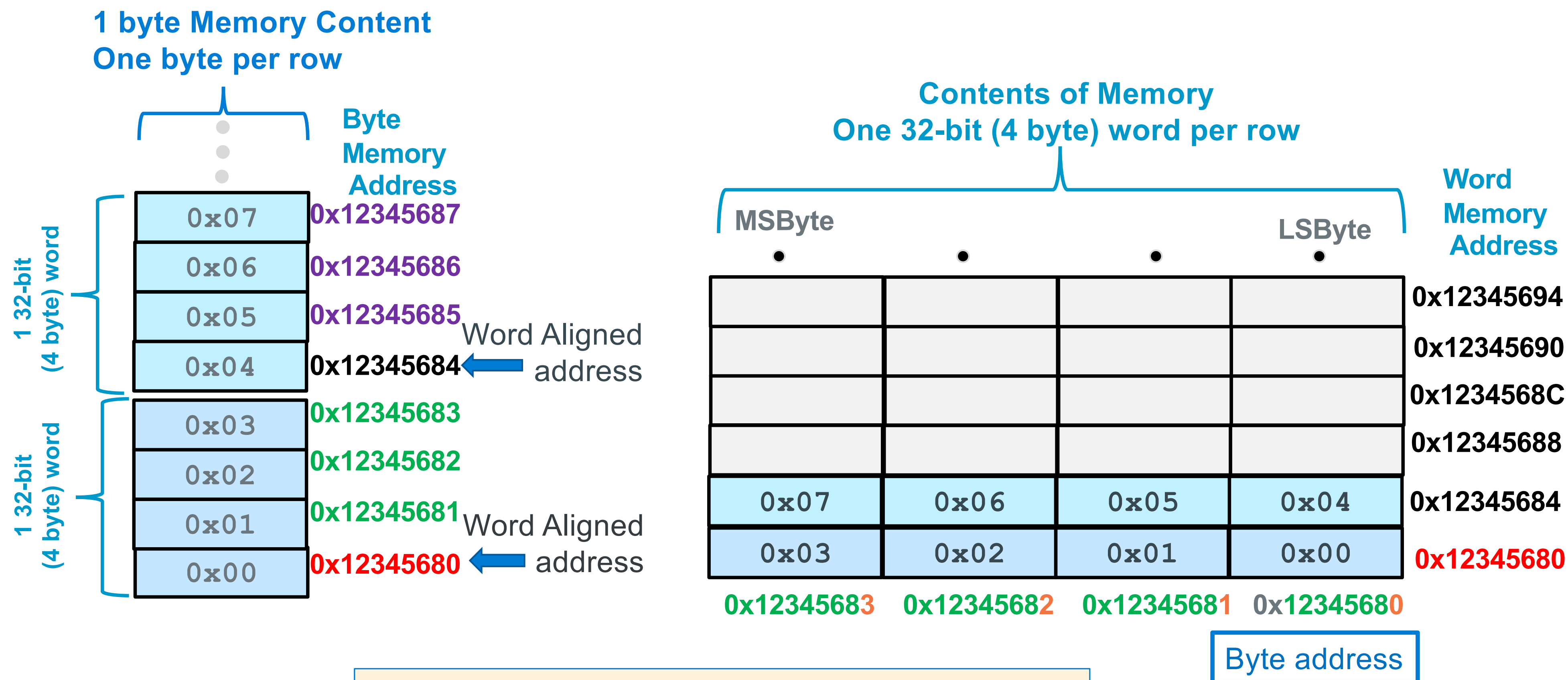
```
Decimal:   12345
Binary:     0011  0000  0011  1001
Hex:          3     0     3     9
```

```
int x = 12345;
// or x = 0x00003039;  // show all 32 bits
```



x

Byte Addressable Memory Shown as 32-bit words



Observation
32-bit aligned addresses
rightmost 2 bits of the address are always 0

x

Scientific Notation Binary

$$+\overbrace{1.01}^{\text{mantissa}}{}_2 \times 2^{-1}$$

exponent

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	x 2^5	= 1.01	x 2^6
1011.1	x 2^5	= 1.0111	x 2^8
0.110	x 2^5	= 1.10	x 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 increased 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 1.202×10^6

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

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)

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:

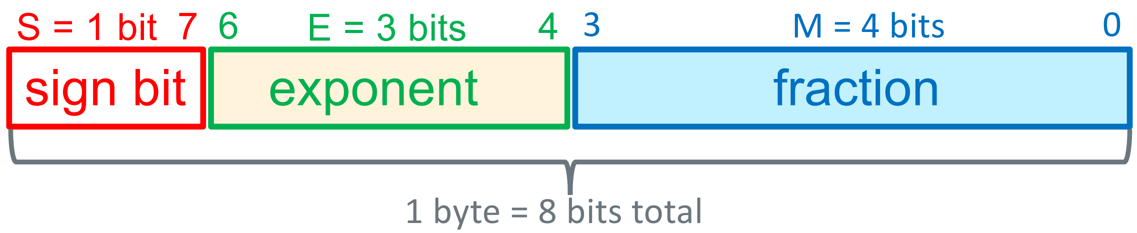
  excess N bias = **(2^{E-1} – 1)** *(this is just one of many bias formulas)*

- **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

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** = **$(2^{E\text{-}1} – 1)$**
  - 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

Scientific Notation Decimal

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

exponent

sign

decimal point

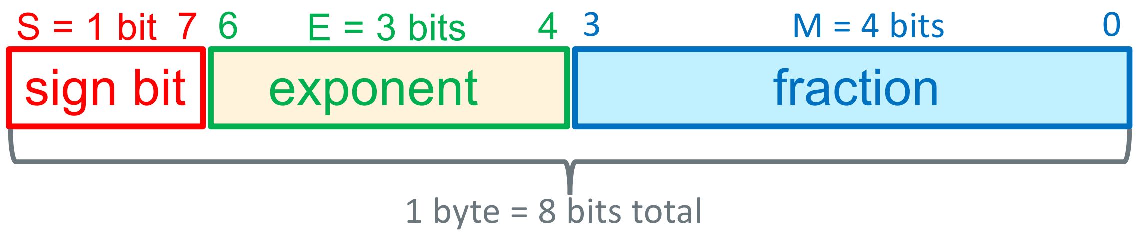
radix (base)

- *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}

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(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 x` $(2^{-2})_{\text{base }10}$

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`

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

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

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

`1.0101`

Step 4: Apply the **unbiased** exponent

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

Step 5: Convert to decimal

$10.101 = 2.625_{base\ 10}$

Step 6: Apply the Sign

$+\ 2.625_{base\ 10}$

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

x

Extra slides

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

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

$\underbrace{1\,100}_{0xc}\;\underbrace{0001}_{0x1}\;\underbrace{0\,101}_{0x5}$ 0110 0000 0000 0000 0000

0xc 0x1 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 x` $(2^2)_{\text{base 10}}$ `= 101.10`

Step 5: Convert to decimal

`101.10 = 5.5`

Step 6: Apply the Sign

`-5.5`

x