

Version 1.02

UCSD CSE 30

Computer Organization and Systems Programming

Numbers Data and Memory

Week 5

Lecture 14

Keith Muller

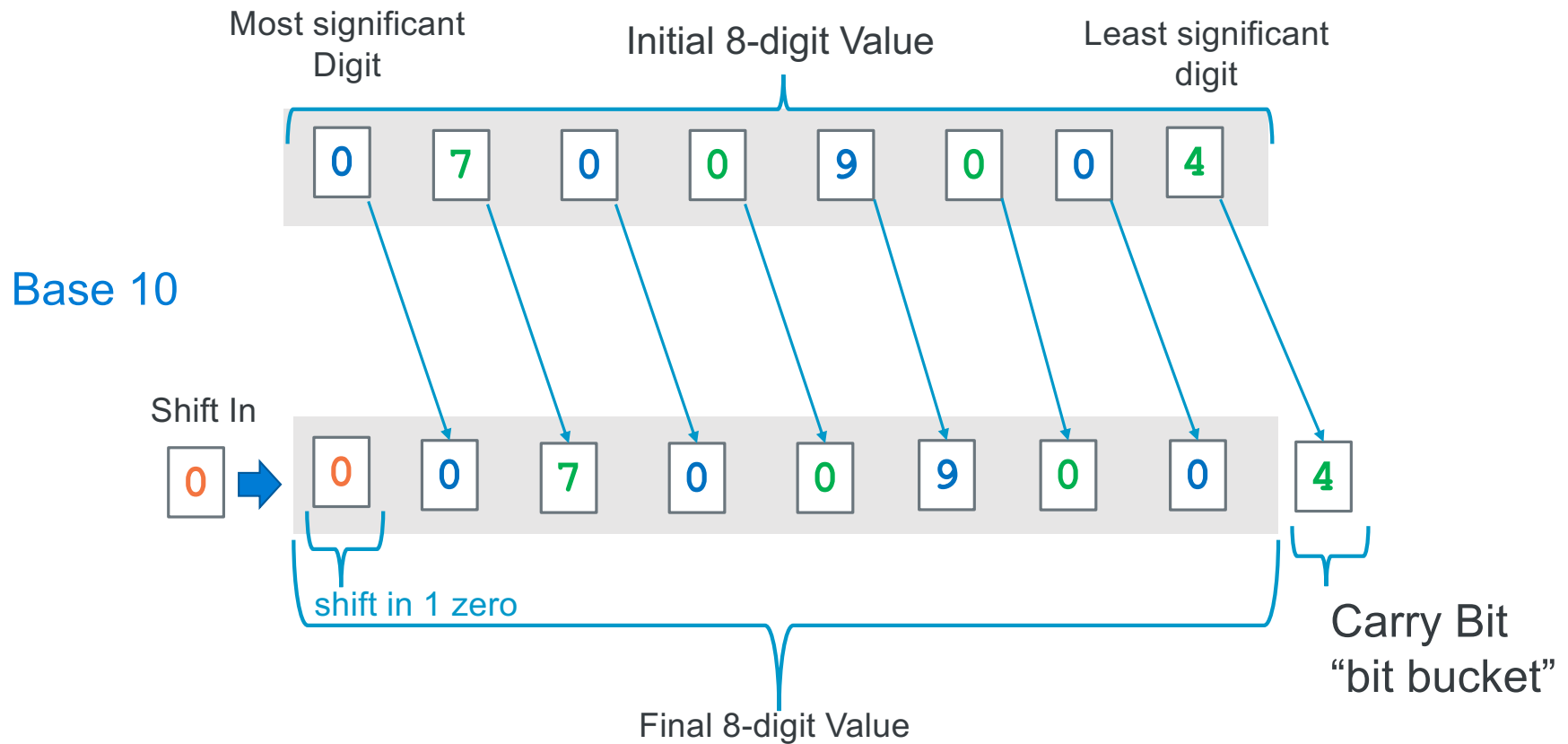
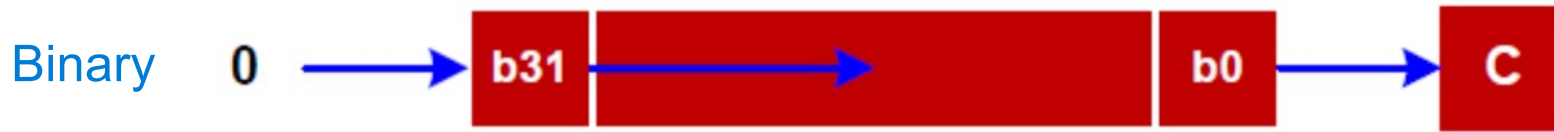


Number Base Overview (as written in C)

- Decimal is base 10, Hexadecimal is base 16, and octal is base 8
- **Octal digits** have 8 values 0 – 7 (written in C as **00** – **07**, careful **073** 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

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



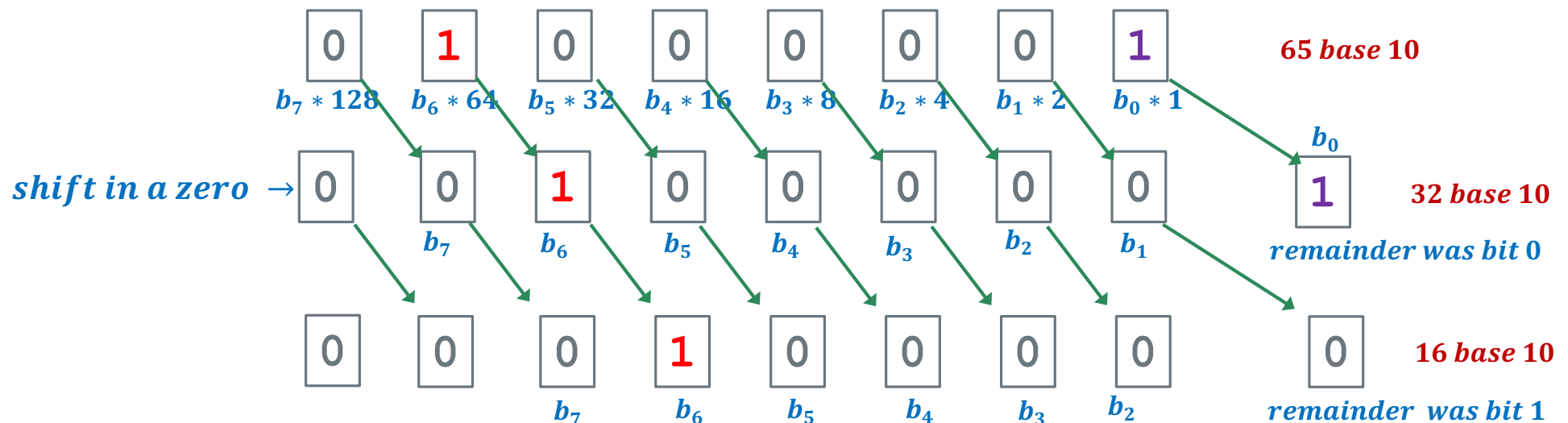
Right Bit Shift: Converts Unsigned Decimal To Unsigned Binary

Algorithmic approach to convert unsigned numbers to binary

- Perform a sequence of **divisions** (**right shift**) to isolate the remainder is a mechanical way to convert decimal to binary

$$\text{Unsigned binary Number} = b_{n-1}2^{N-1} + b_{n-2}2^{N-2} + \dots + b_12^1 + b_02^0$$

$$\text{Unsigned Binary Number} = 2 \times (\dots (2 \times (2 \times b_{n-1} + b_{n-2})) + \dots + b_1) + b_0$$



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

$$249(\text{base } 10) = b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0 = 0b1111001$$

$$11111001 = (1 \times 128) + (1 \times 64) + (1 \times 32) + (1 \times 16) + (1 \times 8) + 1 = 249$$

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

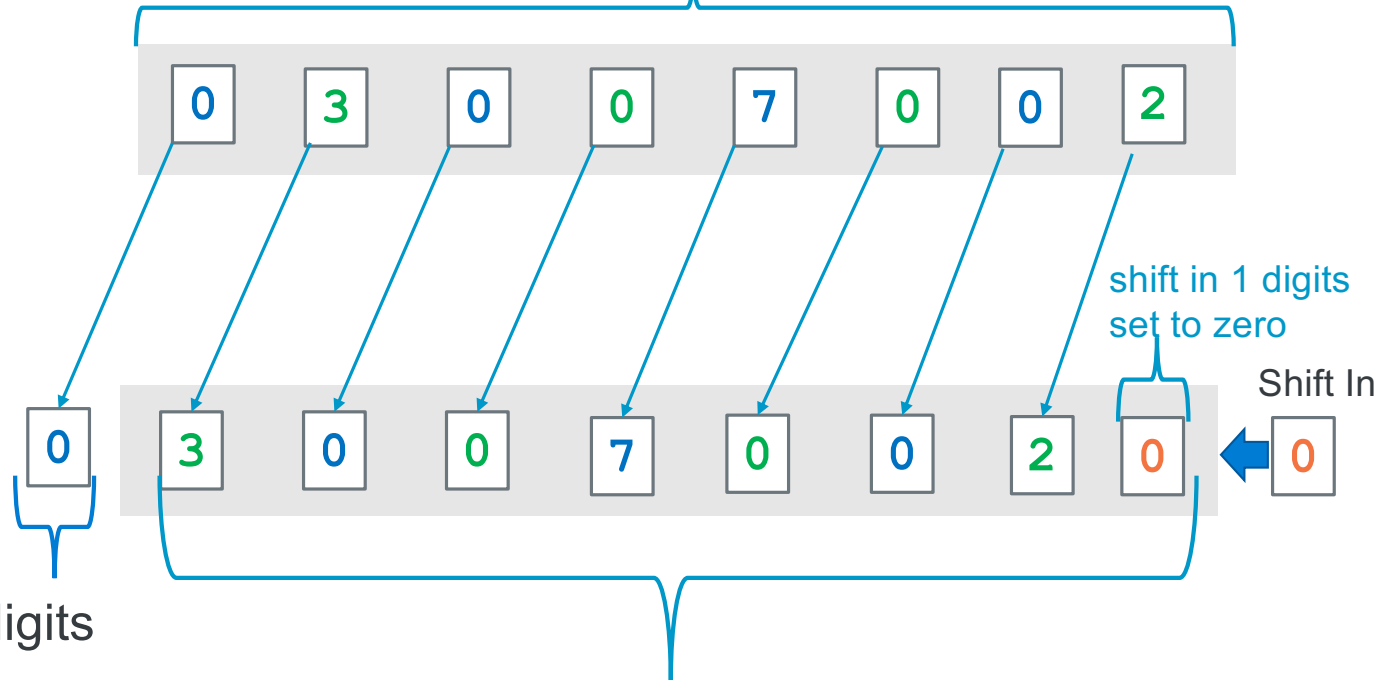


Most significant
Digit

Initial 8-digit Value

Least significant
Digit

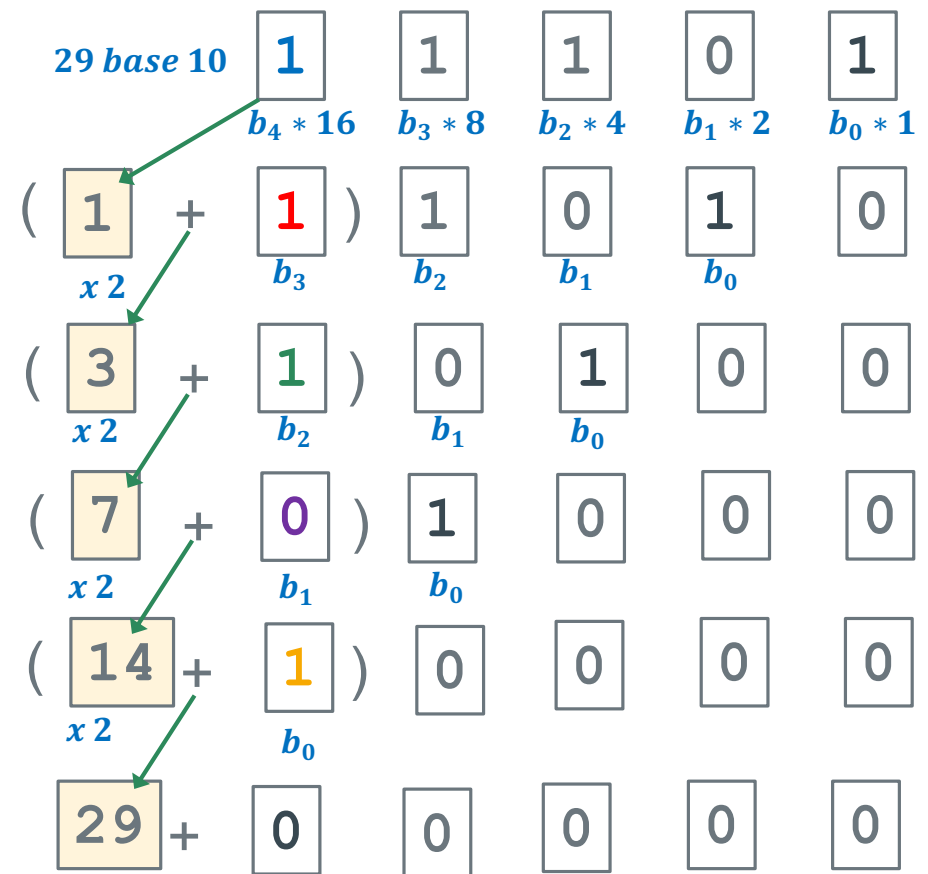
Base 10



Left Bit Shift Converts Unsigned Binary to Unsigned Decimal

$$b_{n-1}2^{N-1} + b_{n-2}2^{N-2} + \dots + b_12^1 + b_02^0$$









- Base conversion via a sequence of n multiplications (left shift) and n additions
- Alternatively, you can memorize and use the positional weights to convert



• $11101_{\text{base } 2} = (1 \times 16) + (1 \times 8) + (1 \times 4) + (0 \times 2) + (1 \times 1) = 29$

Unsigned Binary to Unsigned Decimal Conversion

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

$101_{(\text{base } 10)} = (1 \times 64) + (1 \times 32) + (1 \times 4) + 1$ (checking the conversion)

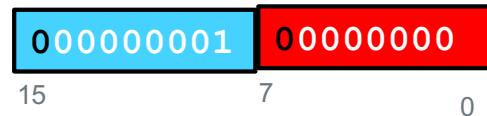
Numbers Are Implemented with a Fixed # of Bits

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

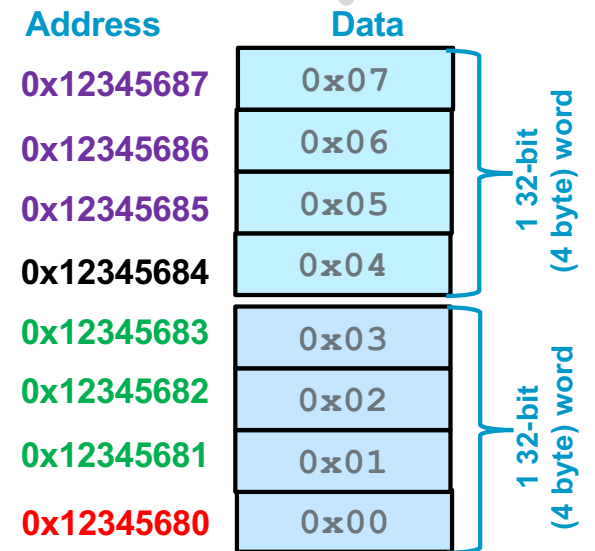
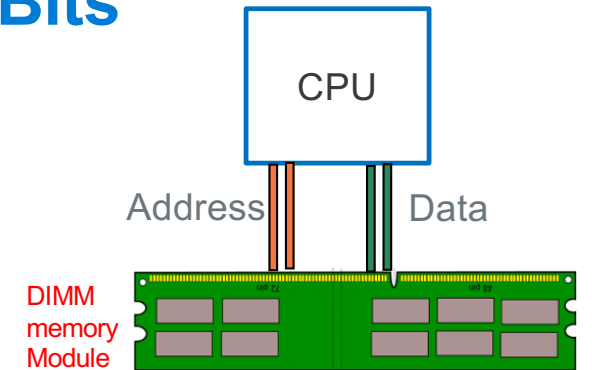
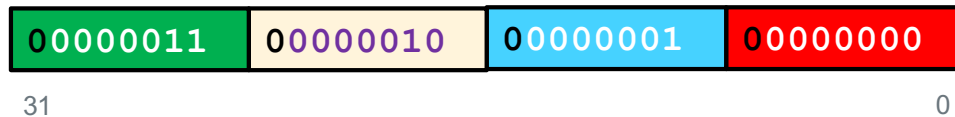
Byte 8-bit integer uses 1 byte



Half Word 16-bit integer uses 2 bytes

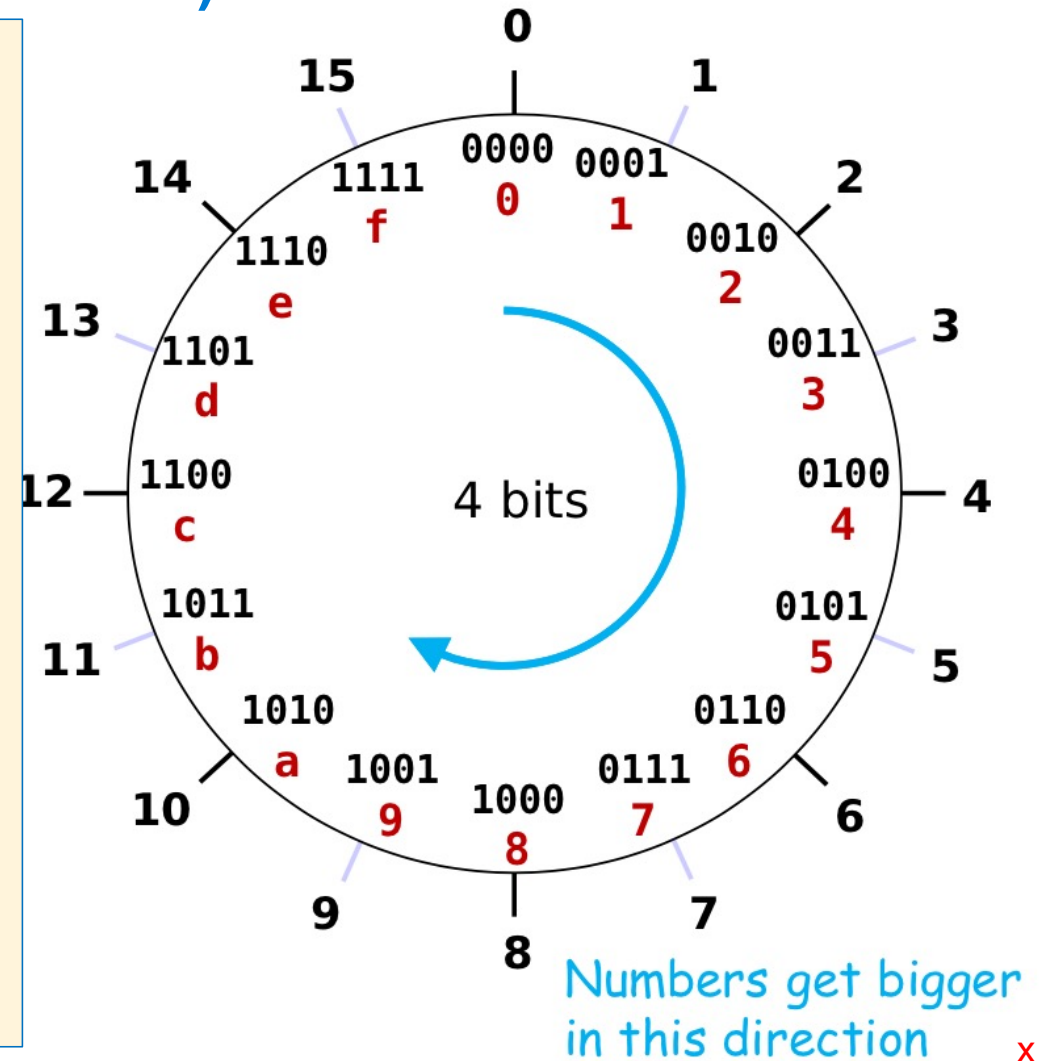


Word 32-bit integer uses 4 bytes



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



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

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

borrows

$$\begin{array}{r} 10100001 \\ - 00110011 \\ \hline \end{array} \quad \begin{array}{r} 161 \\ - 51 \\ \hline \end{array}$$

difference

Be Aware in Binary

$$1 - 1 = 0$$

$$10 - 1 = 1 \text{ base 10: } (2 - 1 = 1)$$

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

borrows		10	10	10	10	10	10	
	0	1	0	1	1	1	0	1
	<hr/>							
	0	0	1	1	0	0	1	1
	<hr/>							
difference	0	1	1	0	1	1	1	0

	161
	<hr/>
	51
	<hr/>
	110

Be Aware in Binary

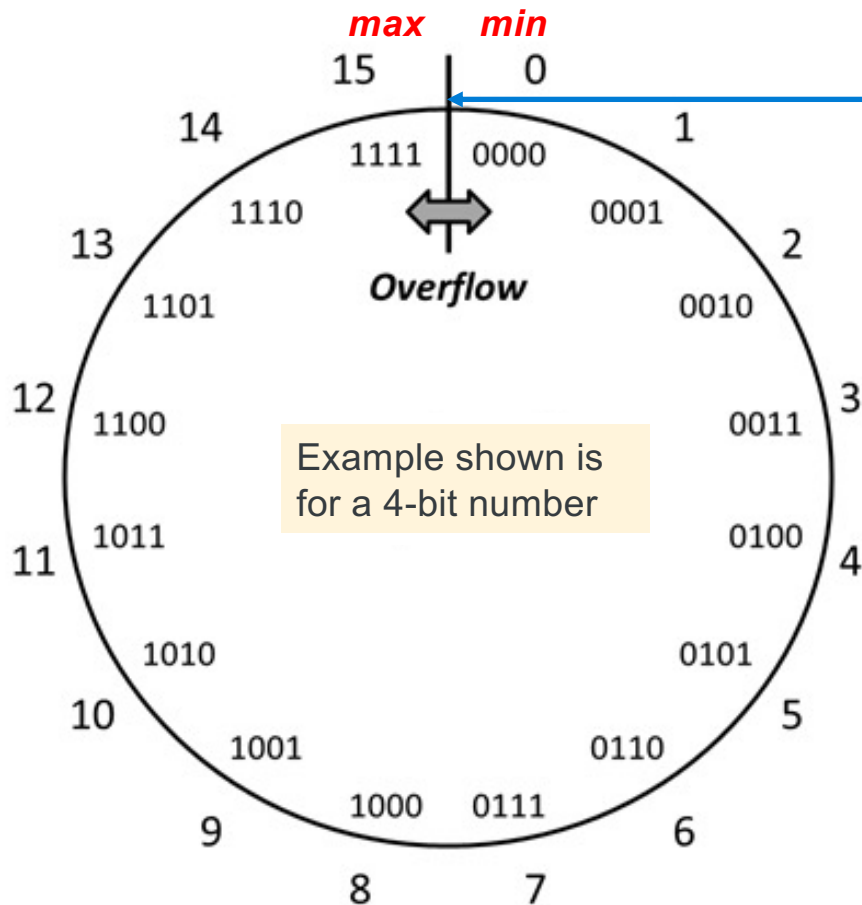
$$1 - 1 = 0$$

$$10 - 1 = 1 \text{ base 10: } (2 - 1 = 1)$$

slide has powerpoint builds

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!....**

Overflow: Unsigned Values 4-bit limit

Addition Overflow: hardware drops carry

$$\begin{array}{r} 15 \\ + 2 \\ \hline 17 \end{array}$$

only 4 bits for
numbers in this
example

carry bit is
always dropped
from result

$$\begin{array}{r} 1111 \\ + 0010 \\ \hline 10001 \\ \text{oops } 1 \end{array}$$

Subtraction Overflow: drops the borrow

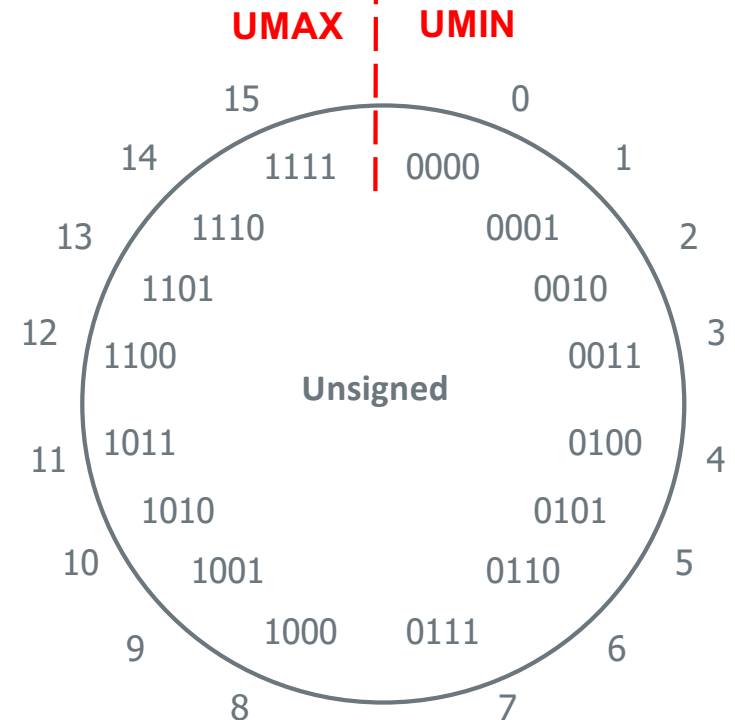
$$\begin{array}{r} 1 \\ - 2 \\ \hline -1 \end{array}$$

only 4 bits for
numbers in this
example

carry bit is
always dropped
from result

$$\begin{array}{r} 10001 \\ - 0010 \\ \hline 1111 \\ \text{oops } 15 \end{array}$$

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



Unsigned Integer Number Overflow: Addition in 8 bits

Carry Bit

carries

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

+

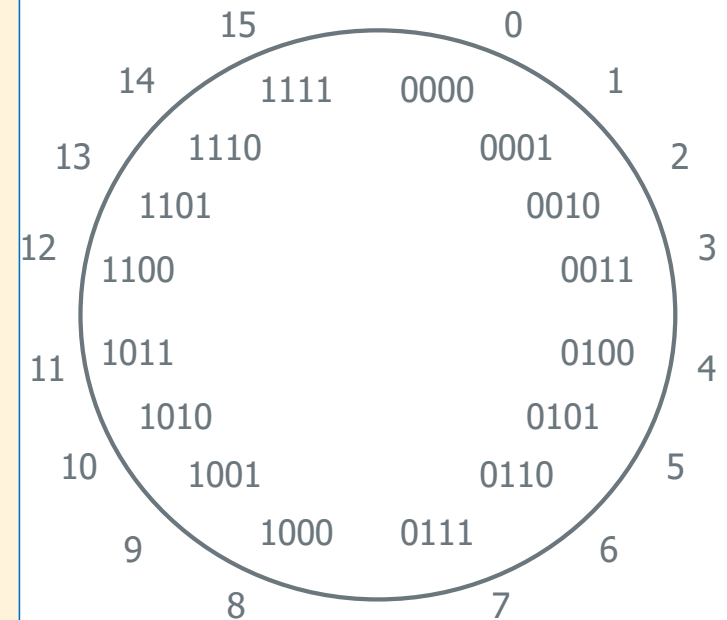
sum

1	1	1	1	1	1	1	1	
1	0	1	0	0	0	0	0	1
0	1	0	1	1	1	1	1	
<hr/>								
0	0	0	0	0	0	0	0	
								161
								95
								<hr/>
								256

Rule: When Carry Bit $\neq 0$, overflow has occurred for unsigned integers!

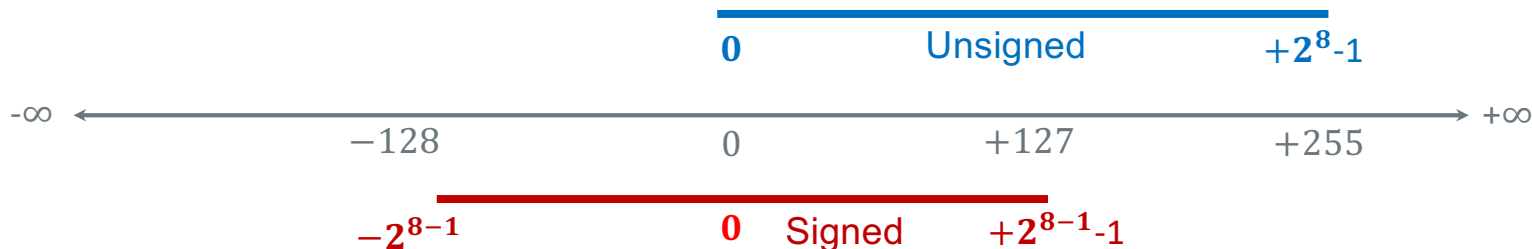
Characteristics of Signed Numbers

- 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
 - $\text{number} + (- \text{representation of number}) = 0$
- 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



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)
- 2^n distinct bit patterns to encode positive and negative values
- Unsigned values:** $0 \dots 2^n - 1$ ← -1 comes from counting 0 as a "positive" number
- Signed values:** $-2^{n-1} \dots 2^{n-1} - 1$ (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

Version 1.02

UCSD CSE 30

Computer Organization and Systems Programming

Numbers Data and Memory

Week 5

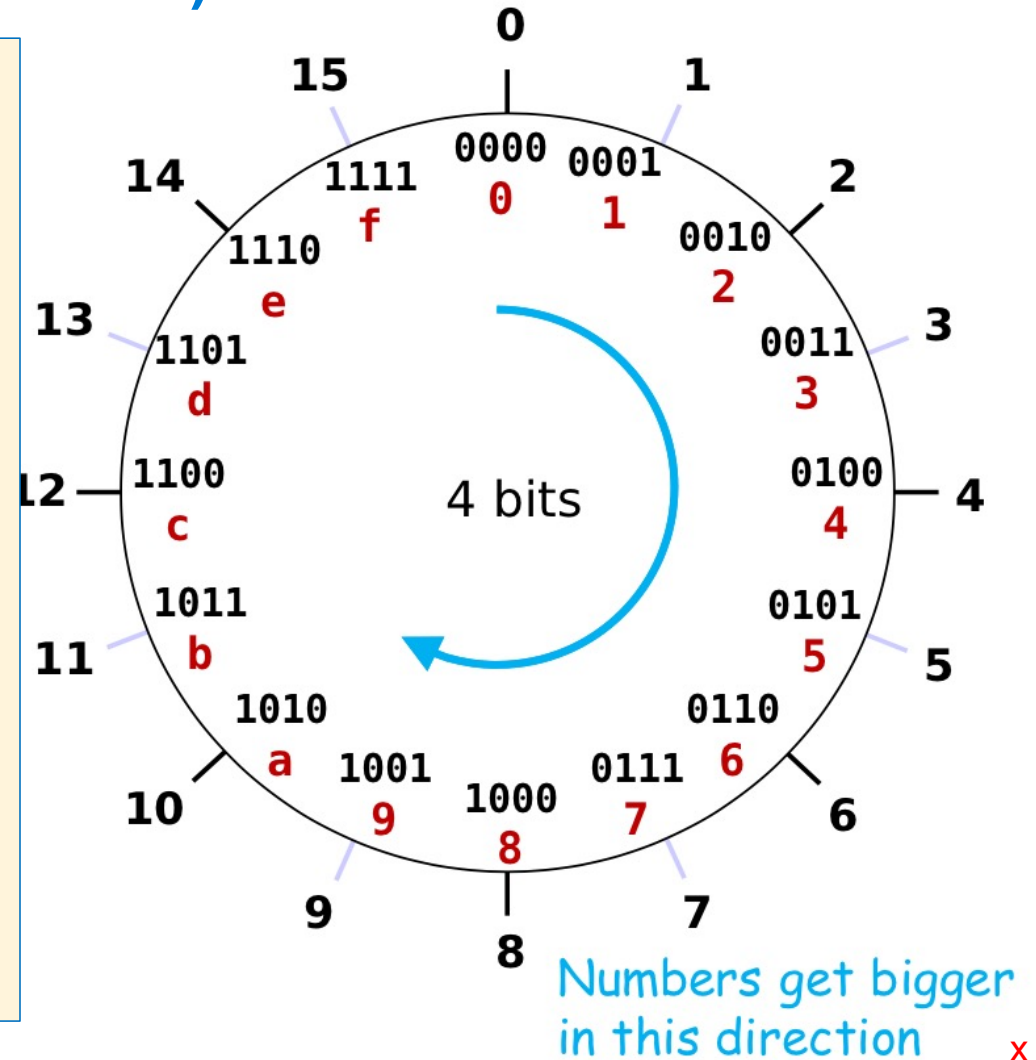
Lecture 15

Keith Muller



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



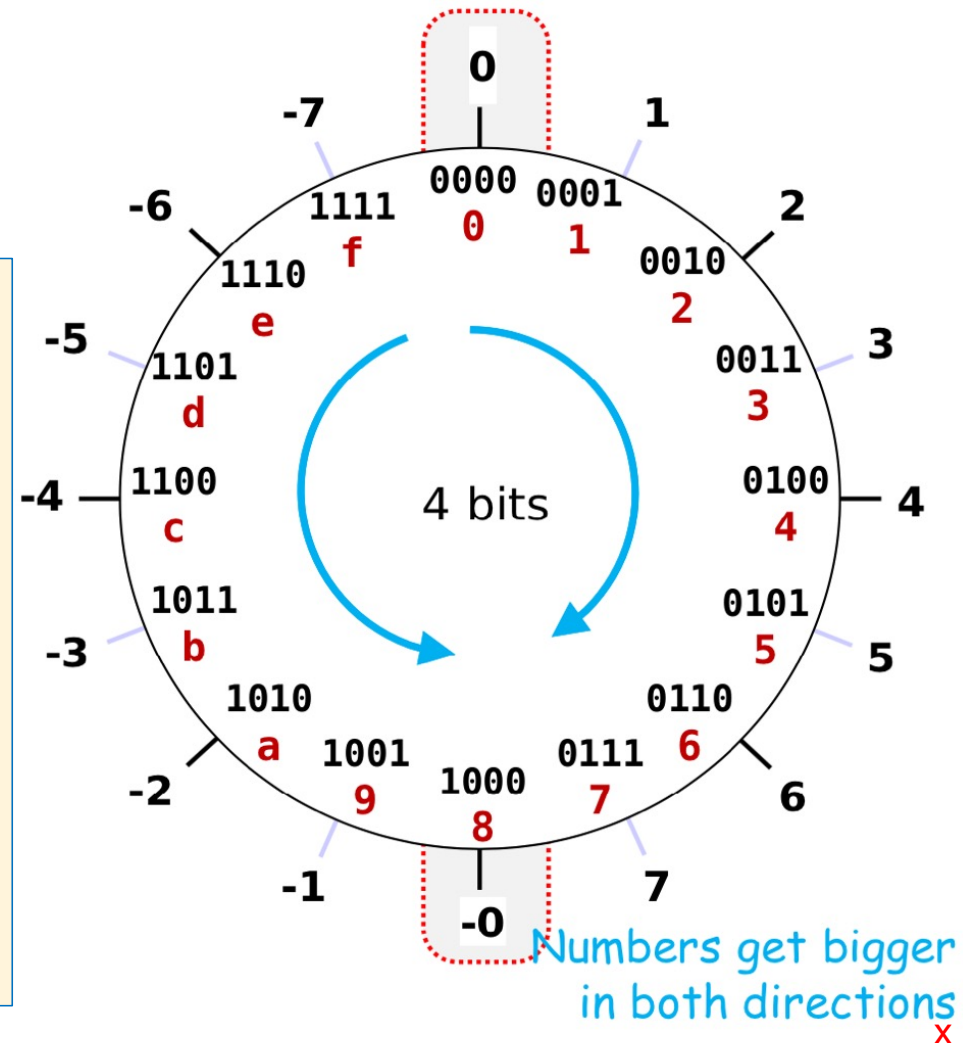
Negative Integer Numbers: Sign + Magnitude Method



- 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)

$\begin{array}{r} 4 \\ - 3 \\ \hline 1 \end{array}$	$\begin{array}{r} 0100 \\ - 0011 \\ \hline 0001 \end{array}$ <p>✓</p>	$\begin{array}{r} 4 \\ + -3 \\ \hline -7 \end{array}$	$\begin{array}{r} 0100 \\ + 1011 \\ \hline 1111 \end{array}$ <p>✗</p>
---	---	---	---

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



Signed Magnitude Examples (Sign bit is always MSB)

Examples (4 bits):

0 110
positive 6

1 011
negative 3

1 000 = -0?

1 001 = -1

1 010 = -2

1 011 = -3

1 100 = -4

1 101 = -5

1 110 = -6

1 111 = -7

0 000 = 0?

0 001 = 1

0 010 = 2

0 011 = 3

0 100 = 4

0 101 = 5

0 110 = 6

0 111 = 7

0 00000000
positive 0

1 0001100
negative 12

Examples Using Hex notation (8 bits):

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

0x7F = 0b01111111 is positive (+127₁₀)

0x85 = 0b10000101 is negative (-5₁₀)

0x80 = 0b10000000 is negative... also zero

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

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:

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

- 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

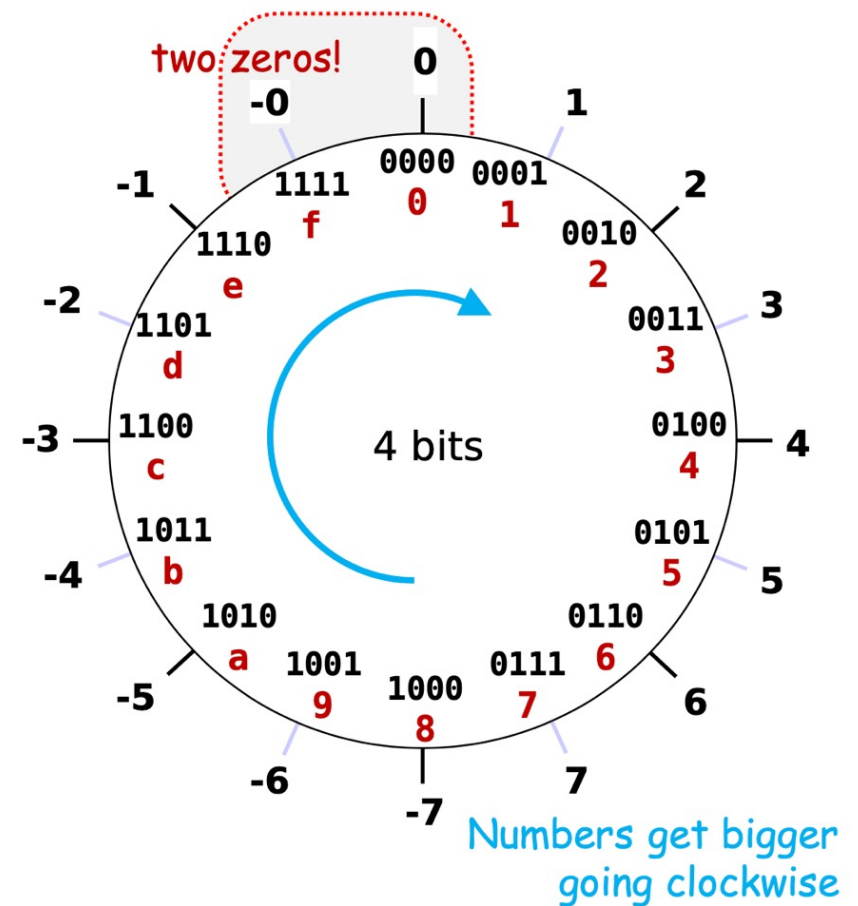
actual	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8
bias	+7	+7	+7	+7	+7	+7	+7	+7	+7	+7	+7	+7	+7	+7	+7	+7
bias encoded	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

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**

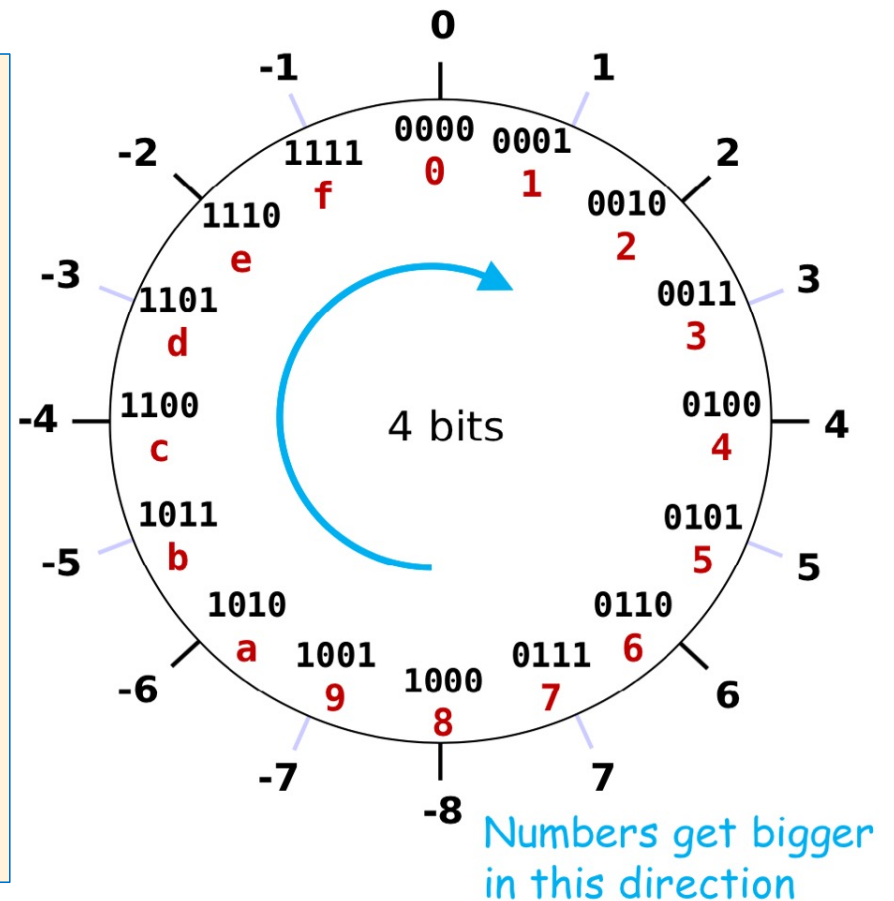
Number	+	-
0	0000	1111
1	0001	1110
2	0010	1101
3	0011	1100
4	0100	1011
5	0101	1010
6	0110	1001
7	0111	1000

- 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

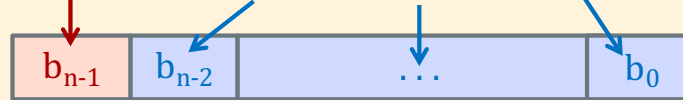
- Positive numbers encoded same as unsigned numbers
- All **negative values** have a **one in the leftmost bit**
- All **positive values** have a **zero in the leftmost bit**
 - This implies that 0 is a positive value
- **Only one zero**
- **For n bits, Number range is $-(2^{n-1})$ to $+(2^{n-1} - 1)$**
 - 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!*



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

$$2's\ Comp = -b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \dots + b_12^1 + b_02^0$$

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



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

Another Way to Look at 2's Complement Encoding

- A 2's complement 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

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

Two's Complement → Unsigned for n bits

- **Unsigned Value Range**

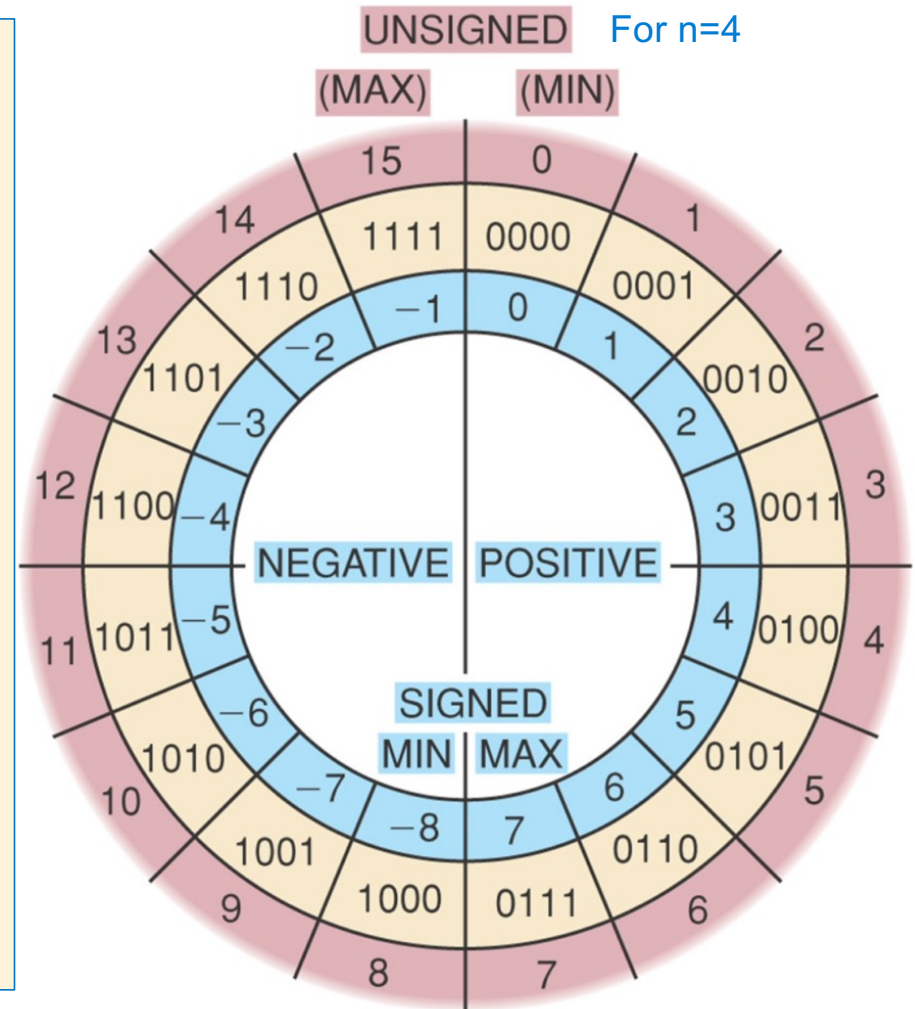
$$\begin{aligned} \text{UMin} &= 0b00\dots00 \\ &= 0 \end{aligned}$$

$$\begin{aligned} \text{UMax} &= 0b11\dots11 \\ &= 2^n - 1 \end{aligned}$$

- **Two's Complement Range**

$$\begin{aligned} \text{SMin} &= 0b10\dots00 \\ &= -2^{n-1} \end{aligned}$$

$$\begin{aligned} \text{SMax} &= 0b01\dots11 \\ &= 2^{n-1} - 1 \end{aligned}$$



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

$$\begin{array}{r}
 7 = 0111 \\
 \downarrow \downarrow \downarrow \downarrow \\
 \text{invert} = 1000 \\
 \text{add } 1 \quad + \quad \underline{\quad 1 \quad} \\
 -7 \quad \quad 1001
 \end{array}$$

$$\begin{array}{r}
 -7 = 1001 \\
 \downarrow \downarrow \downarrow \downarrow \\
 \text{invert} = 0110 \\
 \text{add } 1 \quad + \quad \underline{\quad 1 \quad} \\
 7 \quad \quad 0111
 \end{array}$$

$$-x == \sim x + 1;$$

$$\begin{array}{r}
 7 = \quad \quad 0111 \\
 -7 = \quad + \quad \underline{1001} \\
 \text{(discard carry)} \quad 0000
 \end{array}$$

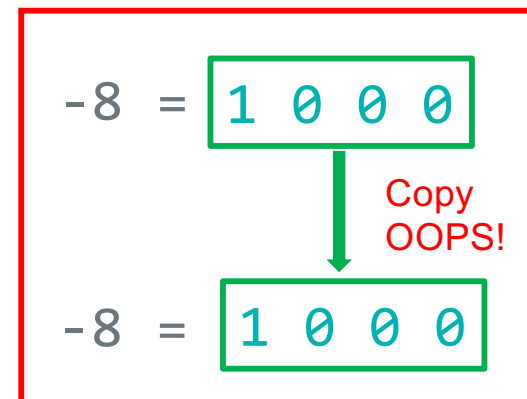
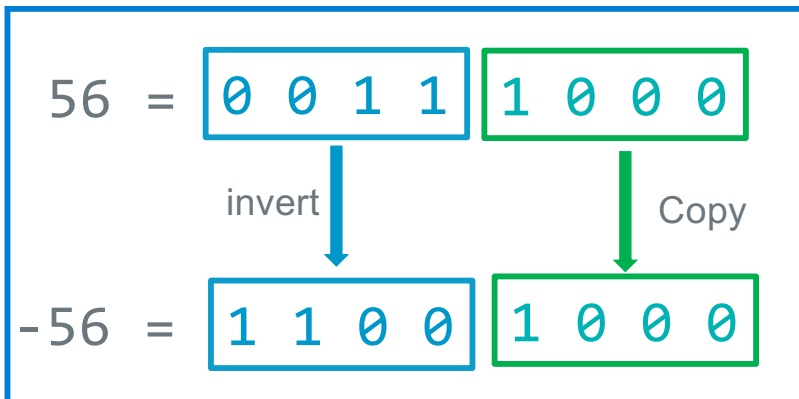
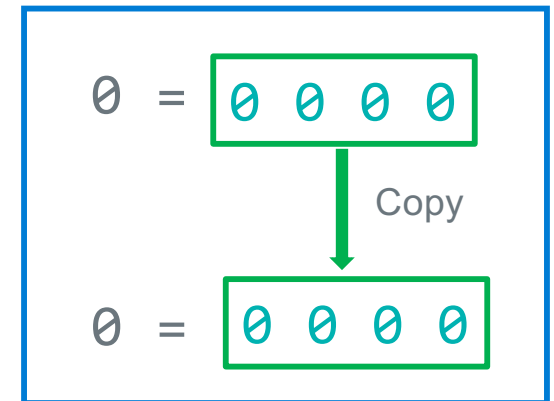
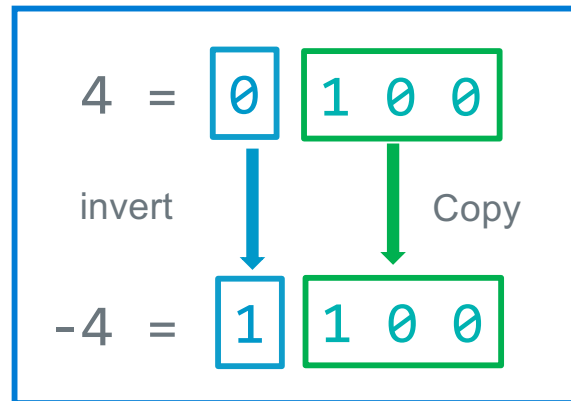
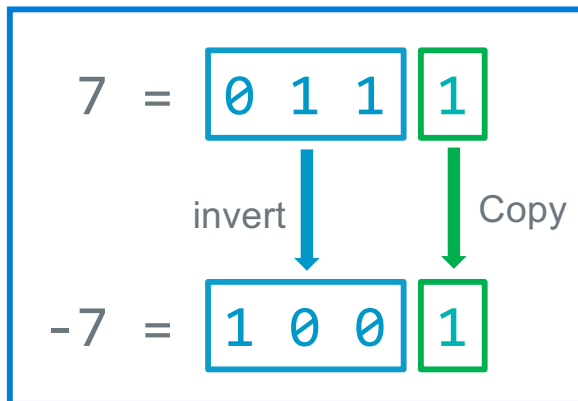
$$\begin{array}{r}
 1 = 0001 \\
 \downarrow \downarrow \downarrow \downarrow \\
 \text{invert} = 1110 \\
 \text{add } 1 \quad + \quad \underline{\quad 1 \quad} \\
 -1 \quad \quad 1111
 \end{array}$$

$$\begin{array}{r}
 -1 = 1111 \\
 \downarrow \downarrow \downarrow \downarrow \\
 \text{invert} = 0000 \\
 \text{add } 1 \quad + \quad \underline{\quad 1 \quad} \\
 1 \quad \quad 0001
 \end{array}$$

$$\begin{array}{r}
 -8 = 1000 \\
 \downarrow \downarrow \downarrow \downarrow \\
 \text{invert} = \underline{0111} \\
 \text{add } 1 \quad + \quad \underline{\quad 1 \quad} \\
 -8 \quad \quad 1000 \text{ oops!}
 \end{array}$$

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



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



Two's Complement to Signed Decimal Conversion - Positive

What is $b_7 \ b_6 \ b_5 \ b_4 \ b_3 \ b_2 \ b_1 \ b_0$
 0 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 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

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

Two's Complement Addition and Subtraction

- **Addition:** just add the two number directly
- **Subtraction:** you can convert to addition: $\text{difference} = \text{minuend} - \text{subtrahend}$
 $\text{difference} = \text{minuend} + 2\text{'s complement}(\text{subtrahend})$

	Count	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
<hr/>									
x + y	=	0	1	0	1	1	1	1	0

$$\begin{array}{r} \mathbf{x} = 0\ 1\ 0\ 1\ 0\ 0\ 1\ 1 \\ \mathbf{y} = 0\ 0\ 0\ 0\ 1\ 0\ 1\ 1 \\ \hline \mathbf{x-y} = 0\ 1\ 0\ 0\ 1\ 0\ 0\ 0 \end{array}$$



2's complement first and then add

$$\begin{array}{r} \mathbf{x} = 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 1 \\ + \ (-\mathbf{y}) = 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 0 \ 1 \\ \hline \mathbf{x} - \mathbf{y} = \mathbf{x} + (-\mathbf{y}) = 0 \ 1 \ 0 \ 0 \ 1 \ 0 \ 0 \ 0 \end{array}$$

Version 1.02

UCSD CSE 30

Computer Organization and Systems Programming

Numbers Data and Memory

Week 6

Lecture 16

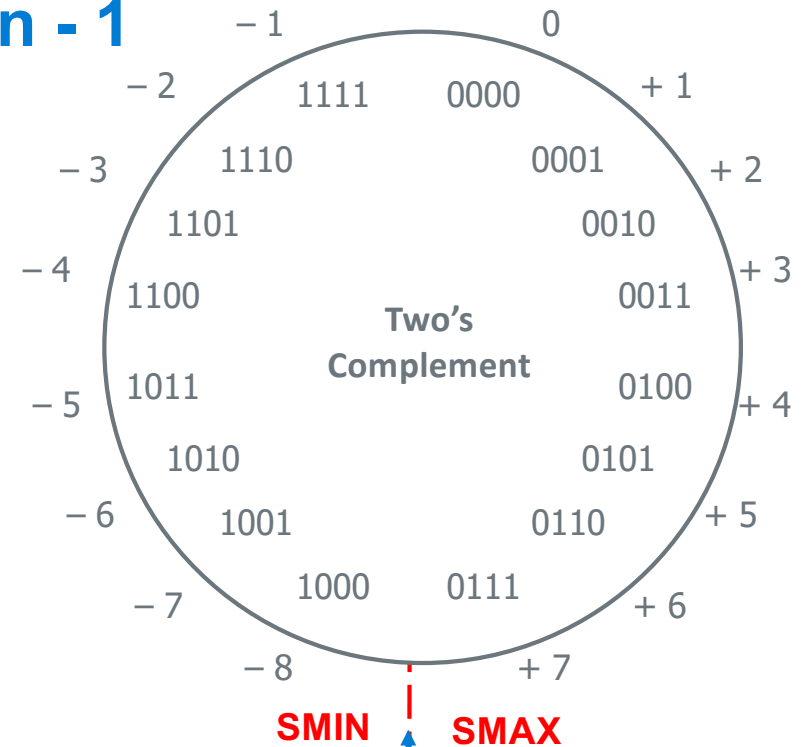
Keith Muller



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

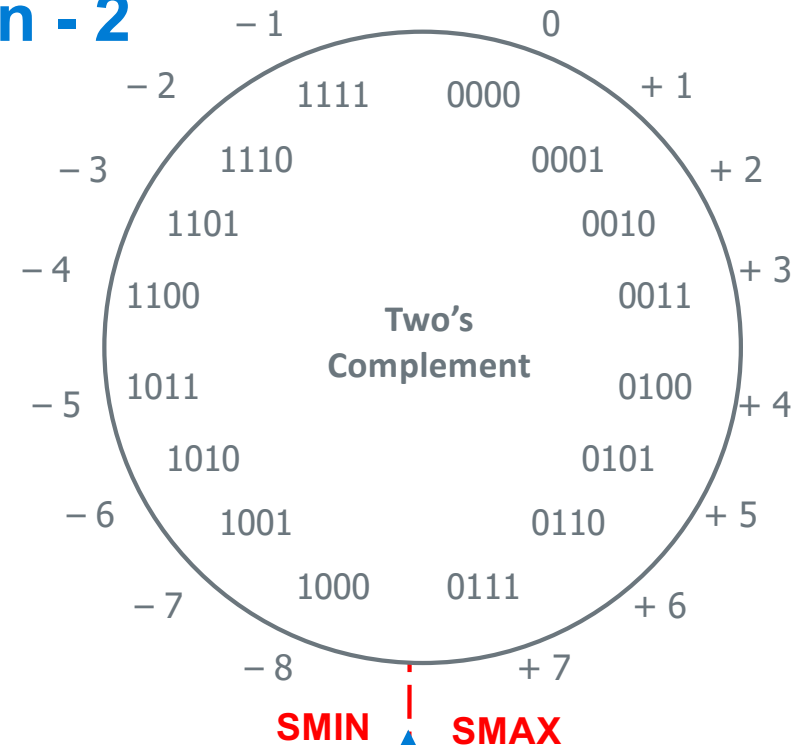
Two's Complement Overflow Detection - 2

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

	Cout	Cin				
	1	0	1	1		
		1	0	0	1	-7
		+	1	0	1	-5
		<hr/>				
		0	1	0	0	+4
						!= -12

carry bit is dropped from result

Result is correct **ONLY** when the **carry into** the sign bit position (MSB) equals the **carry out** of the sign bit position (MSB)



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

Two's Complement Alternative Overflow Detection

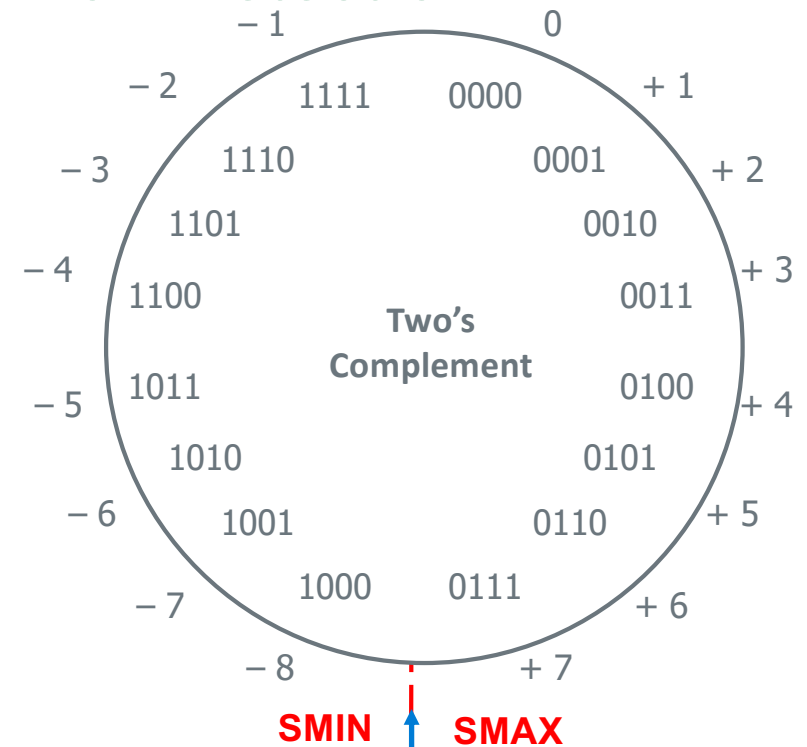
- **Addition:** $(+) + (+) = (-)$ huh?

$$\begin{array}{r}
 6 \\
 + 3 \\
 \hline
 9
 \end{array}
 \qquad
 \begin{array}{r}
 0110 \\
 + 0011 \\
 \hline
 1001 \\
 \text{oops } -7
 \end{array}$$

- **Subtraction:** $(-) + (-) = (+)$ huh?

$$\begin{array}{r}
 -7 \\
 - 3 \\
 \hline
 -10
 \end{array}
 \qquad
 \begin{array}{r}
 1001 \\
 + 1101 \\
 \hline
 0110 \\
 \text{oops } 6
 \end{array}$$

Another Way to look at it for 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

Summary: When Does Overflow Occur

Operand 1
+ Operand 2
Result

Operand 1 Sign	Operand 2 Sign	Is overflow Possible?
+	+	YES
-	-	YES
+	-	NO
-	+	NO

Sign Extension 2's complement number

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

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

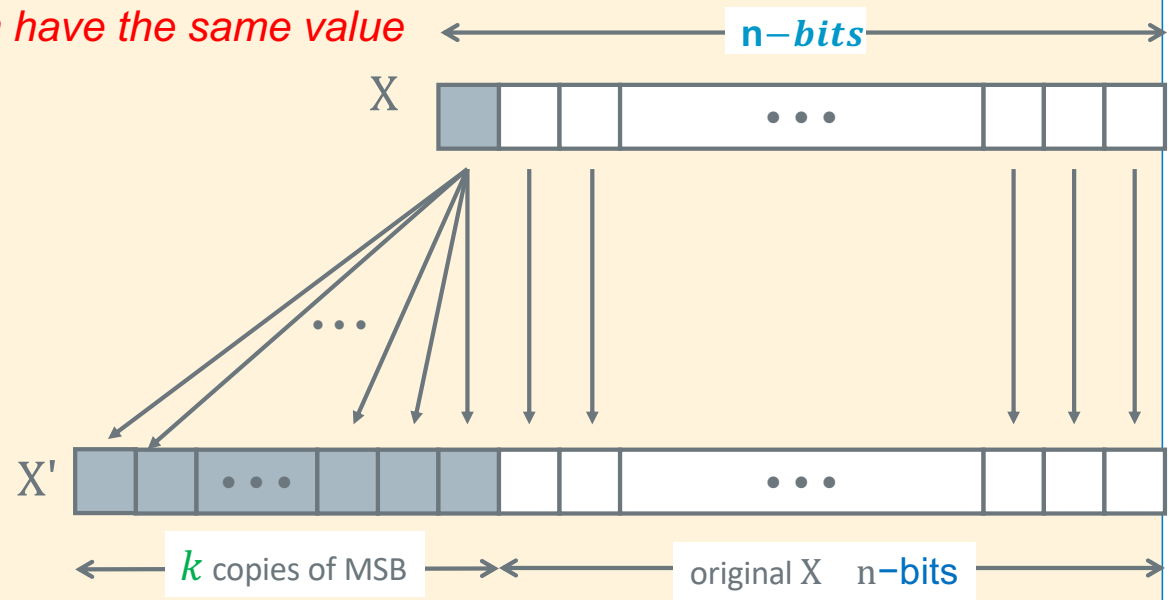
- Sign extension increases the number of bits:** n -bit wide signed integer X , **EXPANDS** to a **wider** n -bit + k -bit signed integer X' where **both have the same value**

Unsigned

- Just add leading zeroes to the left side

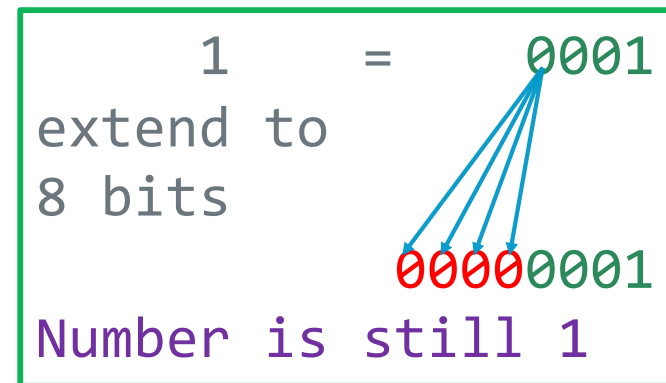
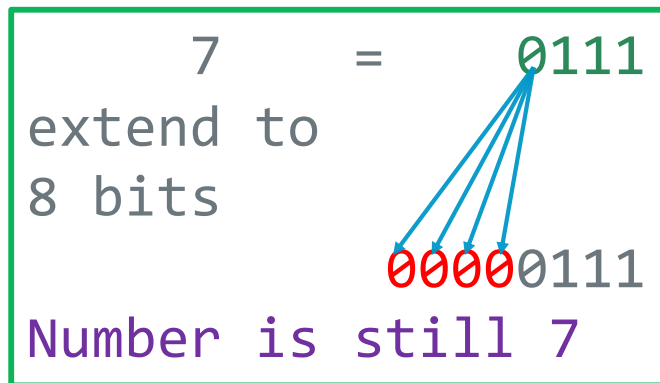
Two's Complement Signed:

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



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

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



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

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

$$\begin{array}{r} 7 = 0111 \\ \quad \downarrow \downarrow \downarrow \downarrow \\ \text{invert} = 1000 \\ \text{add } 1 \quad + \quad \underline{\quad 1 \quad} \\ -7 \quad \quad 1001 \end{array}$$

$$\begin{array}{r} -7 = 1001 \\ \text{extend to} \\ \text{8 bits} \\ \quad \quad \quad \swarrow \downarrow \downarrow \downarrow \downarrow \\ \quad \quad \quad 11111001 \end{array}$$

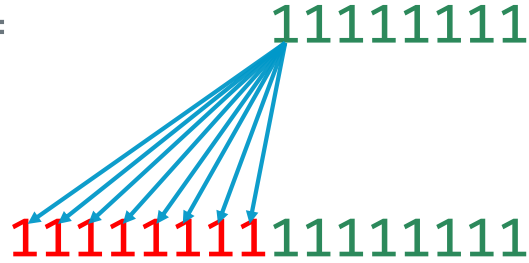


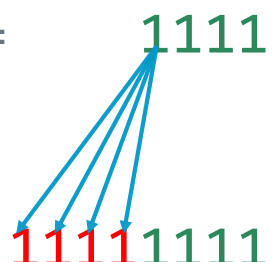
$$\begin{array}{r} 7 = 00000111 \\ \quad \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \downarrow \\ \text{invert} = 11111000 \\ \text{add } 1 \quad + \quad \underline{\quad 1 \quad} \\ -7 \quad \quad 11111001 \end{array}$$

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

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

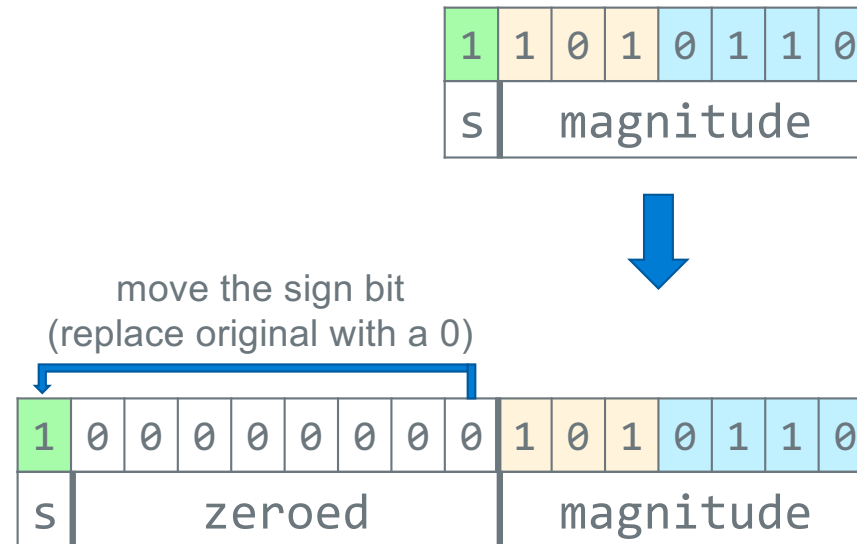
$$\begin{array}{rcl} 1 & = & 0001 \\ & & \downarrow \downarrow \downarrow \downarrow \\ \text{invert} & = & 1110 \\ \text{add } 1 & + & \quad 1 \\ -1 & = & \underline{1111} \end{array}$$

$$\begin{array}{rcl} -1 & = & 11111111 \\ \text{extend to} & & \\ \text{16 bits} & & \end{array}$$


$$\begin{array}{rcl} -1 & = & 1111 \\ \text{extend to} & & \\ \text{8 bits} & & \end{array}$$


Sign Extension Signed Magnitude number

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

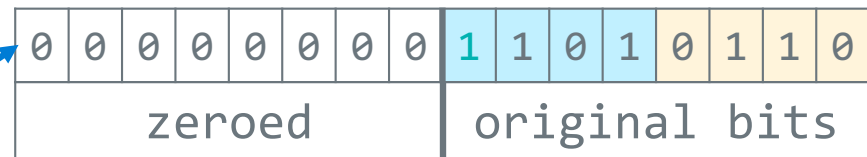


Interpreting and extending with Different representations

How to extend this
bit pattern?

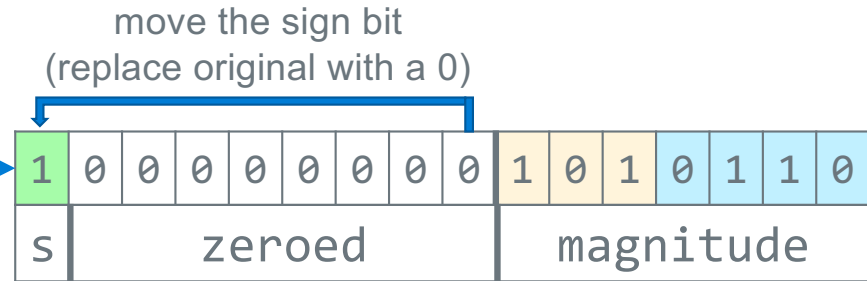
0xd6

unsigned



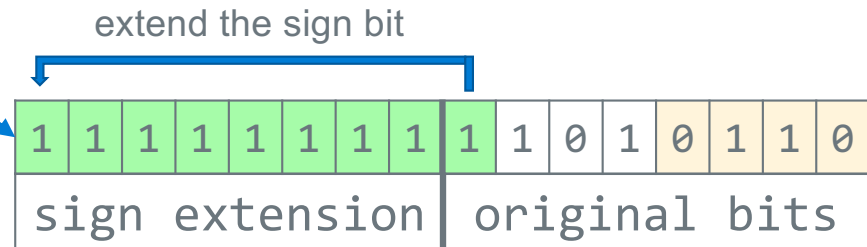
0x00d6

signed
magnitude



0x8056

two's
complement



0xffd6

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)

0b0011

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

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

0b1100

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

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

Shift Operations in C

- n is number of bits to shift a variable x of width w bits
- Shifts by $n < 0$ or $n \geq w$ are *undefined*
- Left shift ($x \ll N$)
 - Shift N bits left, Fill with 0s on right
- In C: behavior of \gg is determined by compiler
 - gcc: it depends on data type of x (signed/unsigned)
- Right shift ($x \gg N$)
 - 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 \ggg and arithmetic shift is \gg



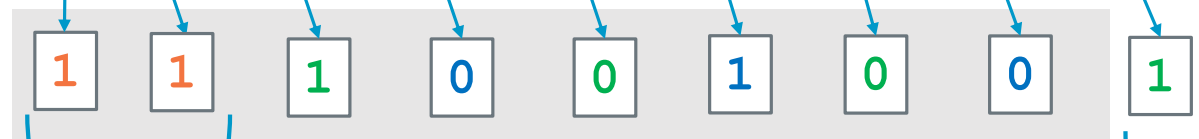
Arithmetic Shift Right 1 Digit = Divide by 2 for 2's Complement Values



Most significant Digit Initial 8-digit Value Least significant digit



Binary
Positive Number



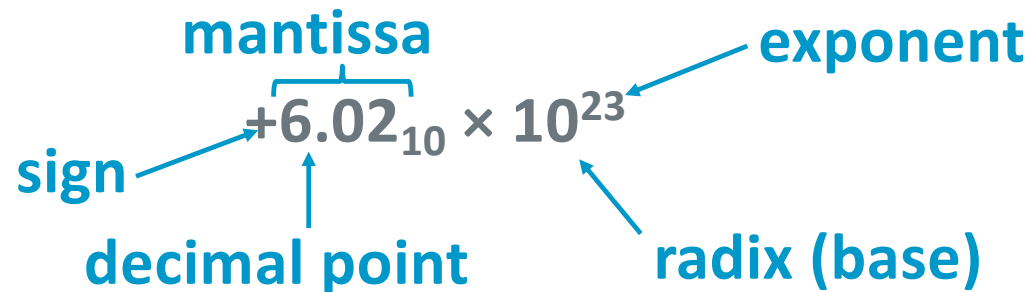
Replicate MSB

Carry Bit
"bit bucket"

Final 8-digit Value

Slides for later in course

Scientific Notation Decimal



- *Scientific Normalized form:*

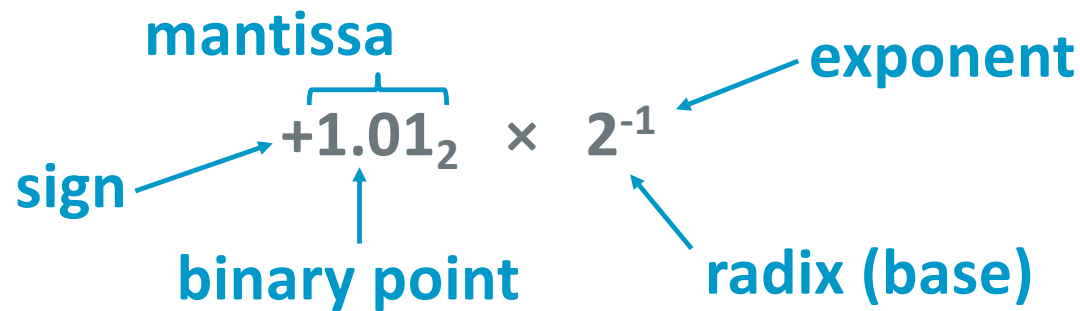
exactly one digit (non-zero) to left of decimal point

- Alternatives to representing 1/1,000,000,000

- **Normalized:** 1.0×10^{-9}

- Not normalized: 0.1×10^{-8} , 10.0×10^{-10}

Scientific Notation Binary



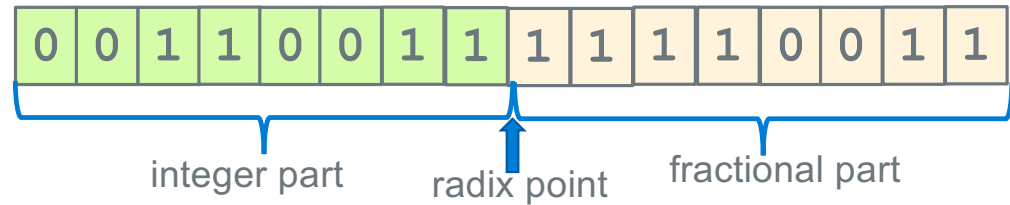
- Computer hardware that supports this is called **floating point hardware** due to the “floating” of the binary point
- Declare such variable in C as `float` (or `double`)

Floating Point Representation

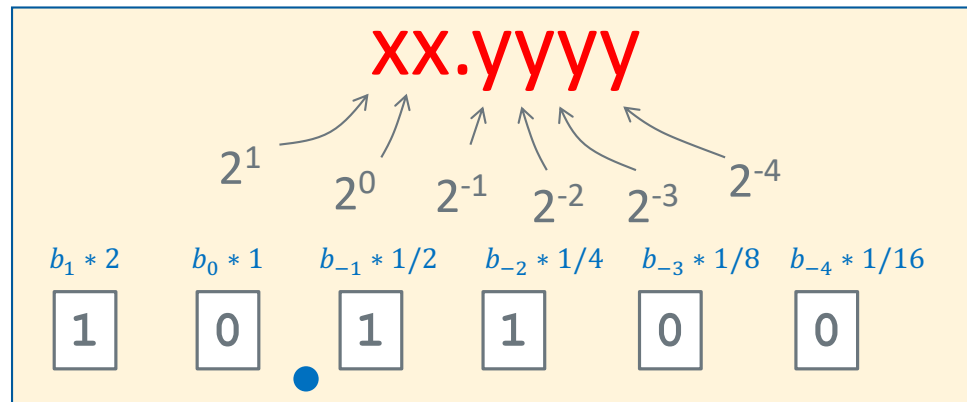
- Analogous to scientific notation
- In Decimal:
 - Not 12000000, but 1.2×10^7 In C: 1.2e7
 - Not 0.0000012, but 1.2×10^{-6} In C: 1.2e-6
- In Binary:
 - Not 11000.000, but 1.1×2^4
 - Not 0.000101, but 1.01×2^{-4}

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: $10.1010_2 = 1 \times 2^1 + 1 \times 2^{-1} + 1 \times 2^{-3} = 2.625_{10}$



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: $1101.001_2 = 1.101001_2 \times 2^3$

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

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

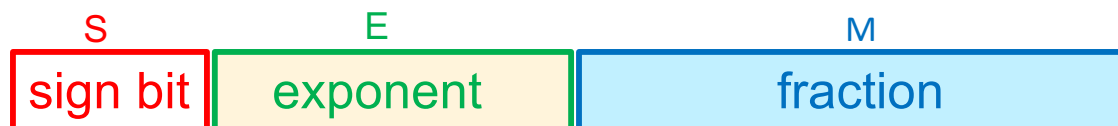
- Adjust x to always be in the format **1.XXXXXXXXXX...** (**fraction is normalized**)
- Fraction portion ONLY **encodes** what is *to the right* of the decimal point
- “Hidden bit” allows number to have **One additional digit for increased precision**

Fraction encoding is **1.[FRACTION BINARY DIGITS]**

Floating Point Numbers: Implementation Approach

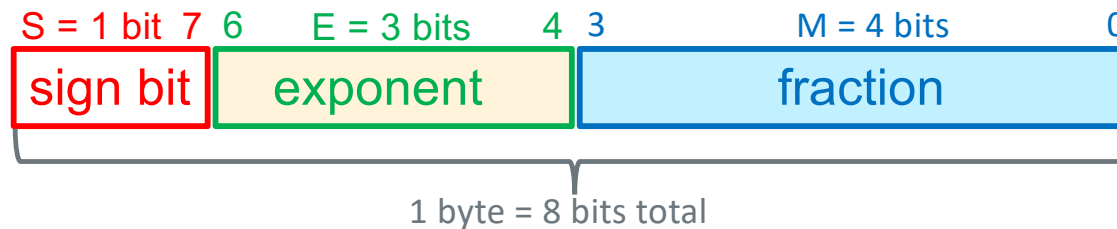
- Supports a wide range of numbers
- Flexible “floating” decimal point
- Represent scientific notation numbers like 1.202×10^6

$$(-1)^S M 2^E$$



- **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
- **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

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



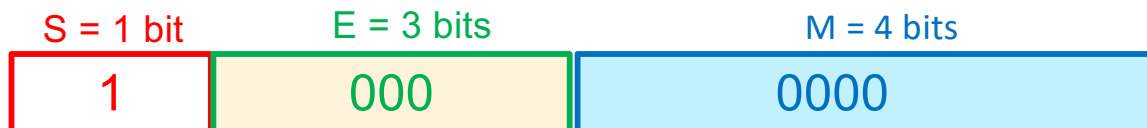
0.0 Special case in this simple model
we do not put back the “hidden bit”



Smallest Non-zero Positive
 $0.00\textcolor{blue}{10001} = \textcolor{blue}{1}/8 + 1/128 = 0.1328125$ base 10



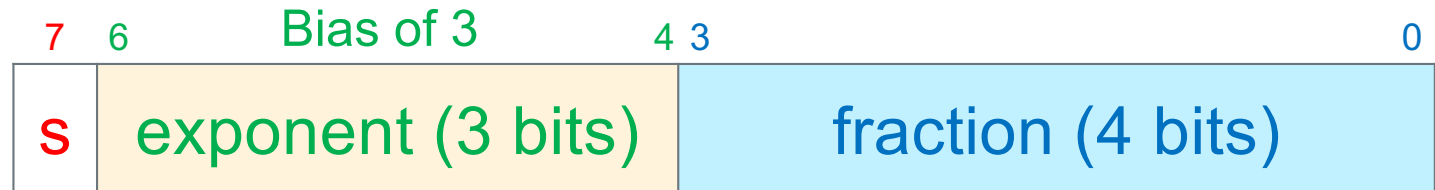
Largest Positive/Negative
 $\textcolor{blue}{1}.\textcolor{blue}{1111} \times 2^4 = \textcolor{blue}{11111} = 31$ base 10



Smallest (closest to zero) Number
 $\textcolor{blue}{1}.0000 \times 2^{-3} = 0.00\textcolor{blue}{1000} = \textcolor{blue}{1}/8 = -0.125$ base 10

Note: Orange is hidden bit added back

Decimal to Float



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

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

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

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

$$1.1000$$

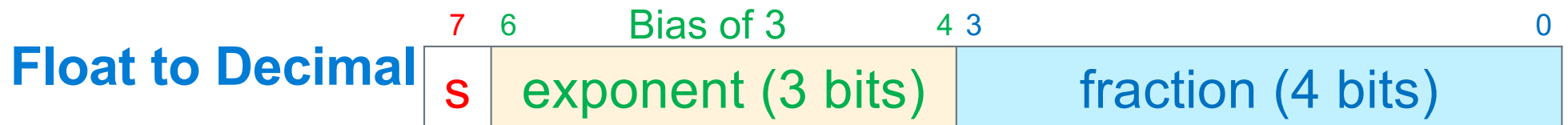
Step 4: Sign bit

positive sign bit is 0

negative sign bit is 1

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

$$= 0x98$$



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} - bias of 3₁₀ = 1₁₀ 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} × (2¹)_{base 10} = 10.101

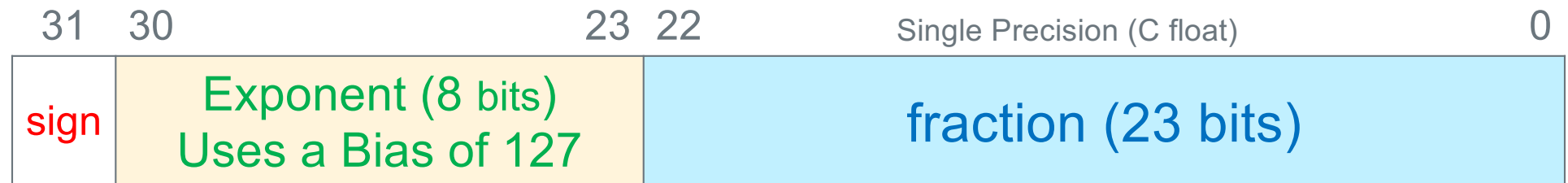
Step 5: Convert to decimal

10.101 = 2.625_{base 10}

Step 6: Apply the Sign

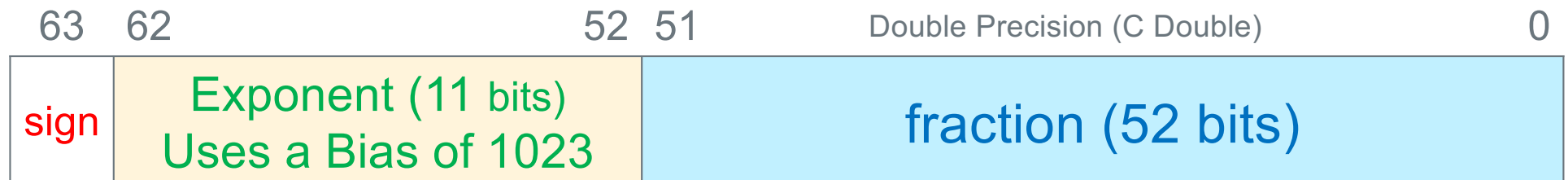
+ 2.625_{base 10}

IEEE “754” Floating Point Double and Single Precision



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

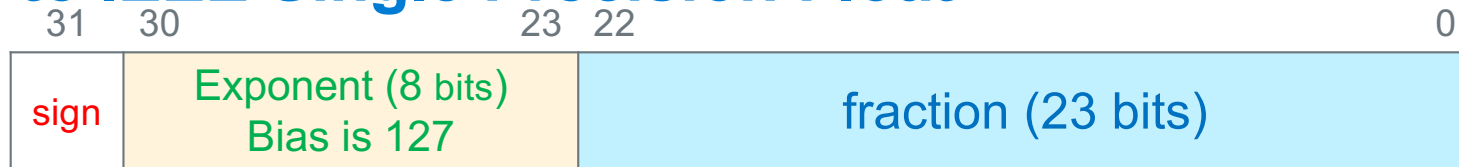
single precision floating point number = $(-1)^s \times 2^{E-127} \times 1.\text{fraction}$



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

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

Decimal to IEEE Single Precision Float



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

$$-13.375(\text{decimal}) = 1101.0110$$

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

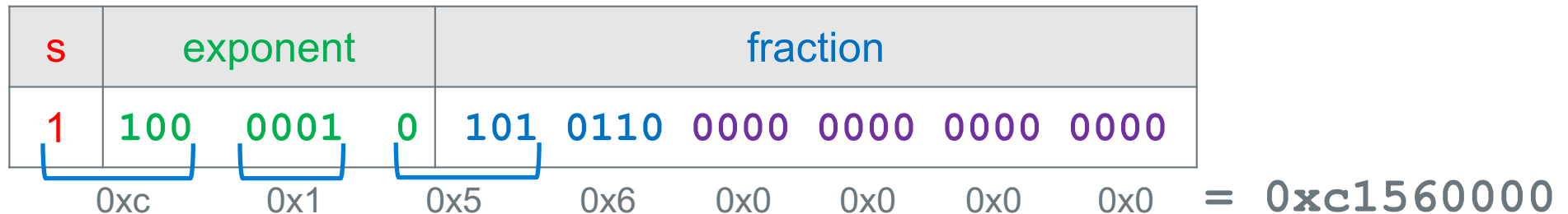
$$1101.0110 = 1.1010110 \times (2^3)_{\text{base } 10}$$

$$3 + \text{bias of } 127 = 130 \text{ for the exponent} = 0b1000 \ 0010$$

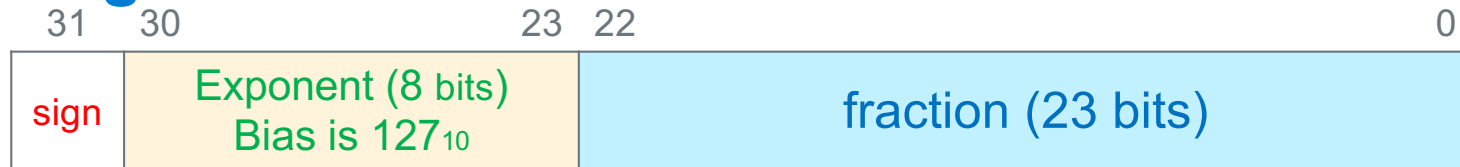
Step 3: Use as many digits that fit to the right of the decimal point in the fractional .xxxx part (0 pad)

$$1.1010110 \ 0000 \ 0000 \ 0000 \ 0000$$

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



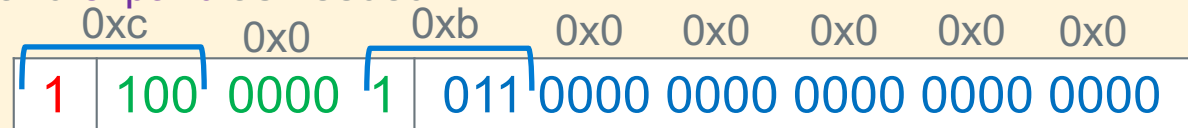
IEEE Single Precision Float to Decimal



Step 1: Break into binary fields and **expand** as needed

$0xc0b00000 =$

Step 2: Find the exponent



$0b10000001 = 129_{\text{base } 10} - \text{bias of } 127_{10} = 2_{10}$ 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

Extra Slides

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

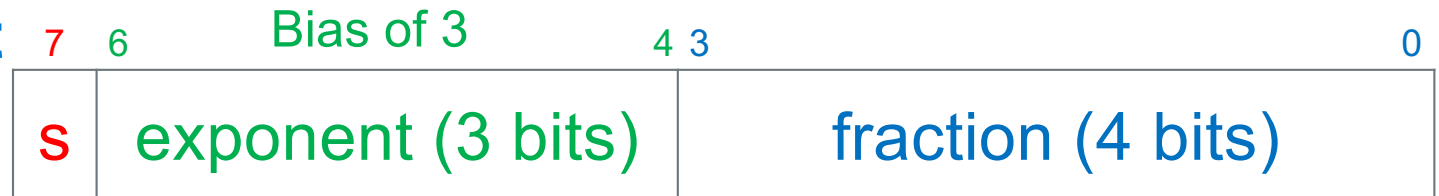
Both operands are positive, but resulting sign is negative
see that cout != cin at the MSB
overflow is detected

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)

Decimal to Float



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

$$6.625 (\text{decimal}) = 0110.1010$$

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

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

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

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

$$1.1010\underline{10} \text{ (we will truncate drop the trailing } \underline{10}, \text{ Real FP use complex rounding approaches)}$$

Step 4: Sign bit

positive sign bit is 0

negative sign bit is 1

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

$$= 0x5a$$