
WELCOME
TO
15-213
18-213
15-513

Bits, Bytes and Integers

14-513/18-613: Introduction to Computer Systems

2nd and 3rd Lectures, May. 20-21, 2020

Announcements

- **CMU Computing and Linux Boot Camp Monday evening during regular class time**
  - A *Quick Start Guide* put together by your hard-working TAs has been posted to Piazza and the course Web site to help you get started until then.

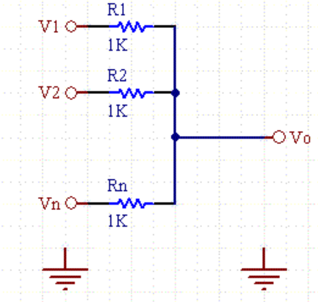
- **Autolab has been created, but I am still configuring it.**
  - You don't need it to start lab 0, which is posted to the Web site
  - It will be available in plenty of time to turn in lab 0 and for the rest of the labs thereafter.

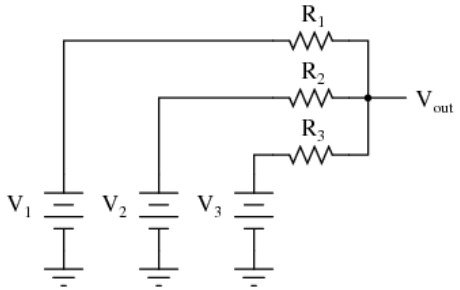
- **Reminder: I've got no control over the waitlist**
  - I've asked the departments and programs to let everyone in
  - I've let the departments and programs know that we have enough TA applicants to hire enough great TAs to fully support the course
  - In the summer, the departments have to work through each student's circumstance one-by-one to do the add. It can take time. A lot of time.

Analog Computers

- **Before digital computers there were analog computers.**

- **Consider a couple of simple analog computers:**
  - A simple circuit can allow one to adjust voltages using variable resistors and measure the output using a volt meter:
  - A simple network of adjustable parallel resistors can allow one to find the average of the inputs.





https://www.daycounter.com/Calculators/Voltage-Summer/Voltage-Summer-Calculator.phtml

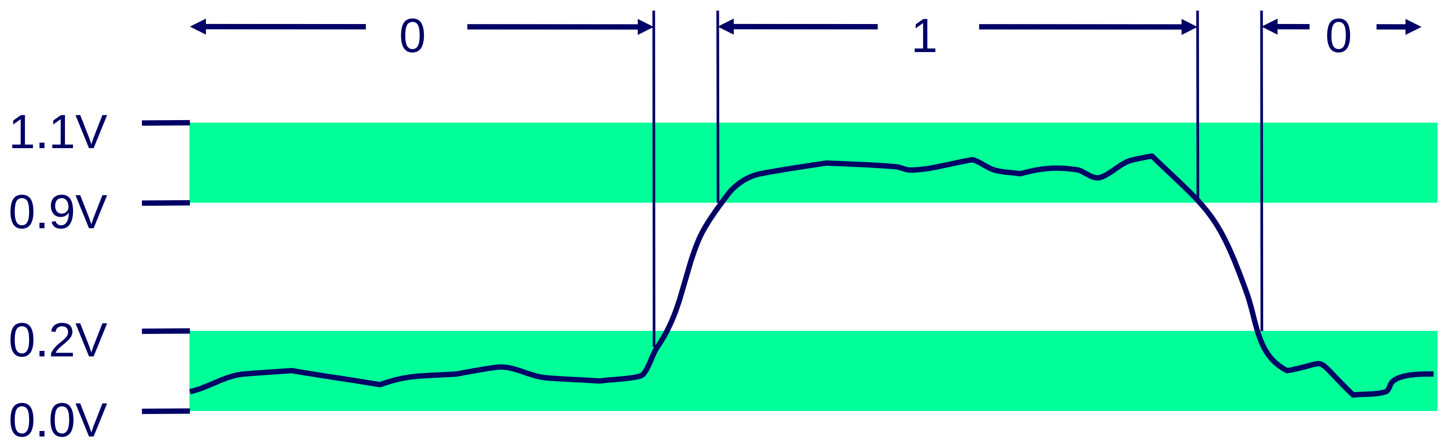
https://www.quora.com/What-is-the-most-basic-voltage-adder-circuit-without-a-transistor-op-amp-and-any-external-supply

The Challenge of Analog Computers

- **All components suffer from tolerances, and noise**
  - Components aren't manufacturer exactly
  - The performance of components varies with the environment and as they age
  - Signals are attenuated and affected by resistance, inductance, capacitance, etc, as they travel through conductors
  - Energy is lost during storage
  - Conductors act as antennas and collect noise
- **These properties mean that nothing is represented the same way over time and space and nothing can be communicated or duplicated or compared exactly**

Needing Less Accuracy, Precision is Better

- **We don't try to measure exactly**
  - We just ask, is it high enough to be "On", or
  - Is it low enough to be "Off".
- **We have two states, so we have a binary, or 2-ary, system.**
  - We represent these states as 0 and 1
- **Now we can easily interpret, communicate, and duplicate signals well enough to know what they mean.**



Binary Representation

- **By encoding/interpreting sets of bits in various ways, we can represent different things:**
  - Operations to be executed by the processor
  - Numbers
  - Enumerable things, such as text characters

- **As long as we can assign it to a discrete number, we can represent it in binary**

Binary Representation: Simple Numbers

- **Binary representation leads to a simple binary, i.e. base-2, numbering system**
  - 0 represents 0
  - 1 represents 1
  - Each "place" represents a power of two, exactly as each place in our usual "base 10", 10-ary numbering system represents a power of 10

Binary Representation: Simple Numbers

- **For example, we can count in binary, a base-2 numbering system**
  - 000, 001, 010, 011, 100, 101, 110, 111, …
    - $000 = 0*2^2 + 0*2^1 + 0*2^0 = 0$ (in decimal)
    - $001 = 0*2^2 + 0*2^1 + 1*2^0 = 1$ (in decimal)
    - $010 = 0*2^2 + 1*2^1 + 0*2^0 = 2$ (in decimal)
    - $011 = 0*2^2 + 1*2^1 + 1*2^0 = 3$ (in decimal)
    - Etc.
- **For reference, consider some base-10 examples:**
    - $000 = 0*10^2 + 0*10^1 + 0*10^0$
    - $001 = 0*10^2 + 0*10^1 + 1*10^0$
    - $357 = 3*10^2 + 5*10^1 + 7*2^0$

Binary Representation:
ASCII Table

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

- 0 (decimal) = 000 (binary)
- 1 (decimal) = 001 (binary)
- 2 (decimal) = 010 (binary)
- Etc.

Encoding Byte Values

- **Bits are very small. It helps to consider groups of them, e.g. Bytes**

- **A Byte = 8 bits**
  - Binary 00000000_2 to 11111111_2
    - Decimal: 0_{10} to 255_{10}

Hexadecimal and Octal

- **Writing out numbers in binary takes too many digits**
- **We want a way to represent numbers more densely such that fewer digits are required**
  - But also such that it is easy to get at the bits that we want
- **Any power-of-two base provides this property**
  - Octal, e.g. base-8, and Decimal, e.g. base-16 are the closest to our familiar base-10.
  - Each has been used by “computer people” over time
  - Hexadecimal is often preferred because it is denser.

Hexadecimal

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

- **Hexadecimal 00_{16} to FF_{16}**
  - Base 16 number representation
  - Use characters ‘0’ to ‘9’ and ‘A’ to ‘F’
- **Consider 1A2B in Hexadecimal:**
  - $1*16^3 + A*16^2 + 2*16^1 + B*16^0$
  - $1*16^3 + 10*16^2 + 2*16^1 + 11*16^0$ = 6699 (decimal)

  - The C Language prefixes hexadecimal numbers with “0x” so they aren’t confused with decimal numbers
  - Write $FA1D37B_{16}$ in C as
    - 0xFA1D37B
    - 0xfa1d37b

15213: $\underbrace{0011}_{3}\ \underbrace{1011}_{B}\ \underbrace{0110}_{6}\ \underbrace{1101}_{D}$

Hexadecimal To Binary

- It is straight-forward to convert a hexadecimal number to binary:
  - Groups of 4 digits represent 16 possibilities, 0-15, i.e. hexadeximal 0-F
- Group the hex digits into groups of 4
  - Start on the left side!
    - If there aren't enough digits, leading 0s can be added on the left, but not on the right.
  - Convert each group of 4 bits into the corresponding hex digit.
  - The concatenation of all of the hex digits is the hex number, because each hex digit represents the same thing as the 4 bits it represents.
- Converting from hex to binary is the reverse process.

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

$$15213\text{:}\ \underbrace{0011}_{3}\ \underbrace{1011}_{B}\ \underbrace{0110}_{6}\ \underbrace{1101}_{D}$$

Common Data Types In the C Language

- Because resources are finite, a fixed amount of memory is usually allocated to data types, including numbers.
  - This amount of memory limits their range and/or precision.
    - We'll talk about that soon
- The table below shows some examples for the C programming Language

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
`char`	1	1	1
`short`	2	2	2
`int`	4	4	4
`long`	4	8	8
`float`	4	4	4
`double`	8	8	8
pointer	4	8	8

Boolean Algebra

- **Developed by George Boole in 19th Century**
  - Algebraic representation of logic
    - Encode "True" as 1 and "False" as 0

And

- **A&B = 1 when both A=1 and B=1**

&	0	1
0	0	0
1	0	1

Or

- **A|B = 1 when either A=1 or B=1**

|	0	1
0	0	1
1	1	1

Not

- **~A = 1 when A=0**

~	
0	1
1	0

Exclusive-Or (Xor)

- **A^B = 1 when either A=1 or B=1, but not both**

^	0	1
0	0	1
1	1	0

General Boolean Algebras

- **Operate on Bit Vectors**
  - Operations applied bitwise

```
  01101001      01101001      01101001
& 01010101    | 01010101    ^ 01010101    ~ 01010101
  01000001      01111101      00111100      10101010
```

- **All of the Properties of Boolean Algebra Apply**

Example: Representing & Manipulating Sets

- **Representation**
  - Width w bit vector represents subsets of {0, …, w–1}
  - $a_j = 1$ if $j \in A$

    - 01101001 { 0, 3, 5, 6 }
    - *76543210*

    - 01010101 { 0, 2, 4, 6 }
    - *76543210*

- **Operations**

&	Intersection	01000001	{ 0, 6 }
|	Union	01111101	{ 0, 2, 3, 4, 5, 6 }
^	Symmetric difference	00111100	{ 2, 3, 4, 5 }
~	Complement	10101010	{ 1, 3, 5, 7 }

Bit-Level Operations in C

- **Operations &, |, ~, ^ Available in C**
  - Apply to any "integral" data type
    - long, int, short, char, unsigned
  - View arguments as bit vectors
  - Arguments applied bit-wise
- **Examples (Char data type)**
  - ~0x41 →
  - ~0x00 →
  - 0x69 & 0x55 →
  - 0x69 | 0x55 →

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

Bit-Level Operations in C

- **Operations &, |, ~, ^ Available in C**
  - Apply to any “integral” data type
    - long, int, short, char, unsigned
  - View arguments as bit vectors
  - Arguments applied bit-wise
- **Examples (Char data type)**
  - ~0x41 → 0xBE
    - ~0100 0001_2 → 1011 1110_2
  - ~0x00 → 0xFF
    - ~0000 0000_2 → 1111 1111_2
  - 0x69 & 0x55 → 0x41
    - 0110 1001_2 & 0101 0101_2 → 0100 0001_2
  - 0x69 | 0x55 → 0x7D
    - 0110 1001_2 | 0101 0101_2 → 0111 1101_2

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

Contrast: Logic Operations in C

- **Contrast to Bit-Level Operators**
  - **Logic Operations: &&, ||, !**
    - View 0 as "False"
    - Anything nonzero as "True"
    - Always return 0 or 1
    - Early termination
- **Examples (char data type)**
  - !0x41 → 0x00
  - !0x00 → 0x01
  - !!0x41→ 0x01

  - 0x69 && 0x55 → 0x01
  - 0x69 || 0x55 → 0x01
  - p && *p (avoids null pointer access)

Watch out for && vs. & (and || vs. |)…
Super common C programming pitfall!

Shift Operations

- **Left Shift: `x << y`**
  - Shift bit-vector **x** left **y** positions
    - – Throw away extra bits on left
    - Fill with 0's on right
- **Right Shift: `x >> y`**
  - Shift bit-vector **x** right **y** positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on left
- **Undefined Behavior**
  - Shift amount < 0 or ≥ word size

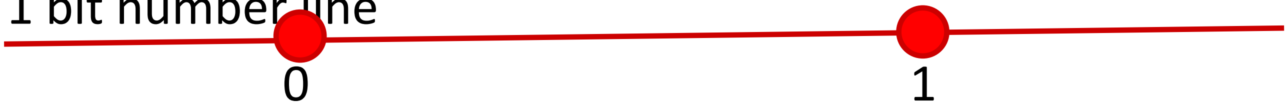
Argument **x**	01100010
<< 3	00010*000*
Log. >> 2	*00*011000
Arith. >> 2	*00*011000

Argument **x**	10100010
<< 3	00010*000*
Log. >> 2	*00*101000
Arith. >> 2	*11*101000

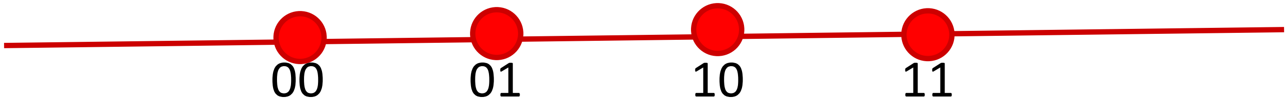
Binary Number Lines

- **In binary, the number of bits in the data type size determines the number of points on the number line.**
  - We can assign the points any meaning we'd like

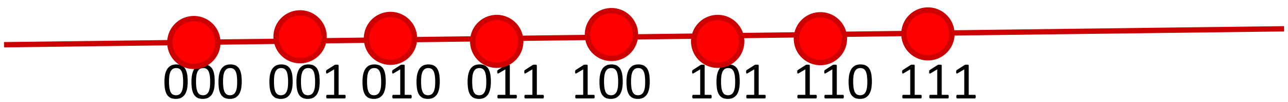
- **Consider the following examples:**
  - 1 bit number line



  - 2 bit number line

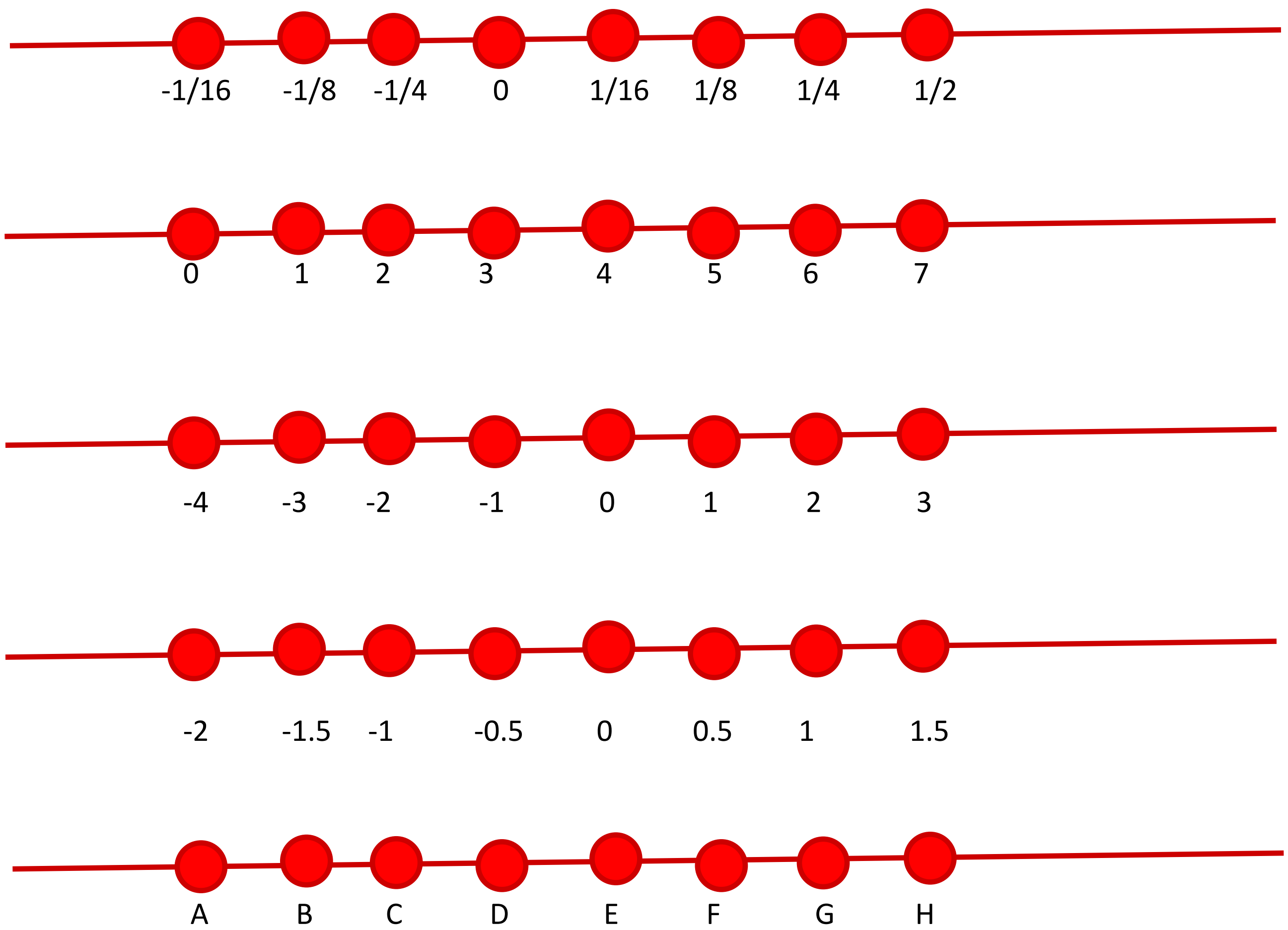


  - 3 bit number line



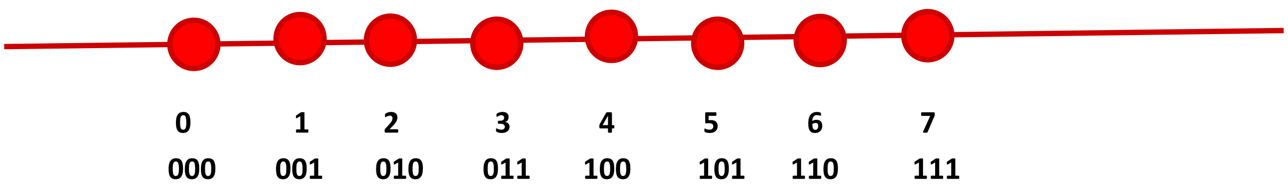
Some Purely Imaginary Examples

- **3 bit number line**



Overflow

- **Let's consider a simple 3 digit number line:**



- **What happens if we add 1 to 7?**
  - In other words, what happens if we add 1 to 111?

- **111+ 001 = 1 000**
  - But, we only get 3 bits – so we lose the leading-1.
  - This is called overflow

- **The result is 000**

Modulus Arithmetic

- **Let's explore this idea of overflow some more**
  - 111 + 001 = 1 000 = 000
  - 111 + 010 = 1 001 = 001
  - 111 + 011 = 1 010 = 010
  - 111 + 100 = 1 011 = 011
  - …
  - 111 + 110 = 1 101 = 101
  - 111 + 111 = 1 110 = 110

- **So, arithmetic "wraps around" when it gets "too positive"**

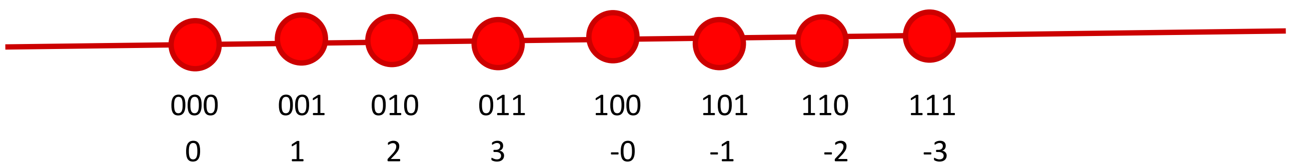
Unsigned and Non-Negative Integers

- **We'll use the term "ints" to mean the finite set of integer numbers that we can represent on a number line enumerated by some fixed number of bits, i.e. *bit width*.**

- **We normally represent unsigned and non-negative int using simple binary as we have already discussed**
  - An "unsigned" int is any int on a number line, e.g. of a data type, that doesn't contain any negative numbers
  - A non-negative number is a number greater than or equal to (>=) 0 on a number line, e.g. of a data type, that does contain negative numbers

How represent negative Numbers?

- **We could use the leading bit as a *sign bit*:**
  - 0 means non-negative
  - 1 means negative



- **This has some benefits**
  - It lets us represent negative and non-negative numbers
  - 0 represents 0

- **It also has some drawbacks**
  - There is a -0, which is the same as 0, except that it is different
  - How to add such numbers 1 + -1 should equal 0
    - But, by simple math, 001 + 101 = 110, which is -2?

A Magic Trick!

- **Let’s just start with three ideas:**
  - 1 should be represented as 1
  - -1 + 1 = 0
  - We want addition to work in the familiar way, with simple rules.
- **We want a situation where “-1” + 1 = 0**
- **Consider a 3 bit number:**
  - 001 + “-1” = 0
  - 001 + 111 = 0
    - Remember 001 + 111 = 1 000, and the leading one is lost to overflow.
- **“-1” = 111**
  - Yep!

Negative Numbers

- **Well, if 111 is -1, what is -2?**
  - -1 - 1
  - 111 – 001 = 110
- **Does that really work?**
  - If it does -2 + 2 = 0
  - 110 + 010 = 1 000 = 000
- **-2 + 5 should be 3, right?**
  - 110 + 101 = 1 011 = 011
- **In general**
  - $-x = -1 - x$

Finding –x the easy way

- **Given a non-negative number in binary, e.g. 5, represented with a fixed bit width, e.g. 4**
  - 0101

- **We can find its negative by flipping each bit and adding 1**
  - 0101 This is 5
  - 1010 This is the "ones complement of 5", e.g. 5 with bits flipped
  - 1011 This is the "twos complement of 5", e.g. 5 with the bits flipped and 1 added
  - 0101 + 1011 = 1 0000 = 0000

- **Because of the fixed with, the "two's complement" of a number can be used as its negative.**

Why Does This Work?

- Consider any number and its complement:
  - 0101
  - 1010
- They are called complements because complementary bits are set. As a result, if they are added, all bits are necessarily set:
  - 0101 + 1010 = 1111
- Adding 1 to the sum of a number and its complement necessarily results in a 0 due to overflow
  - (0101 + 1010) + 1 = 1111 + 1 = 1 0000 = 0000
- And if x + y = 0, y must equal –x
  - So if x + TwosComplement(x) + 1 = 0

Why Does This Work? *Cont.*

- **If x + y = 0**
  - y must equal –x

- **So if x + (TwosComplement(x) + 1) = 0**
  - TwosComplement(x) + 1 must equal –x

- **Another way of looking at it:**
  - if x + (TwosComplement(x) + 1) = 0
  - x + TwosComplement(x) = -1
  - x = -1 - TwosComplement(x)
  - -x = 1 + TwosComplement(x)

Two-complement Encoding Example (Cont.)

```
x =      15213: 00111011 01101101
y =     -15213: 11000100 10010011
```

Weight	15213		-15213	
1	1	1	1	1
2	0	0	1	2
4	1	4	0	0
8	1	8	0	0
16	0	0	1	16
32	1	32	0	0
64	1	64	0	0
128	0	0	1	128
256	1	256	0	0
512	1	512	0	0
1024	0	0	1	1024
2048	1	2048	0	0
4096	1	4096	0	0
8192	1	8192	0	0
16384	0	0	1	16384
-32768	0	0	1	-32768
Sum		**15213**		**-15213**

Negation: Complement & Increment

- **Negate through complement and increase**

  `~x + 1 == -x`

- **Example**
  - Observation: **`~x + x == 1111…111 == -1`**

	x	1	0	0	1	1	1	0	1
+	~x	0	1	1	0	0	0	1	0
	-1	1	1	1	1	1	1	1	1

x = 15213

	Decimal	Hex	Binary
`x`	15213	`3B 6D`	`00111011 01101101`
`~x`	-15214	`C4 92`	`11000100 10010010`
`~x+1`	-15213	`C4 93`	`11000100 10010011`
`y`	-15213	`C4 93`	`11000100 10010011`

Complement & Increment Examples

x = 0

	Decimal	Hex	Binary
0	0	00 00	00000000 00000000
~0	-1	FF FF	11111111 11111111
~0+1	0	00 00	00000000 00000000

x = Tmin (The most negative two's complement number)

	Decimal	Hex	Binary
x	-32768	80 00	10000000 00000000
~x	32767	7F FF	01111111 11111111
~x+1	-32768	80 00	10000000 00000000

Canonical counter example

Encoding Integers: Dense Form

Unsigned

$$B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$$

Two's Complement

$$B2T(X) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i$$

Sign Bit

```
short int x =  15213;
short int y = -15213;
```

- **C does not mandate using two's complement**
  - But, most machines do, and we will assume so
- **C `short` 2 bytes long**

	Decimal	Hex	Binary
`x`	15213	`3B 6D`	`00111011 01101101`
`y`	-15213	`C4 93`	`11000100 10010011`

- **Sign Bit**
  - For 2's complement, most significant bit indicates sign
    - 0 for nonnegative
    - 1 for negative

Numeric Ranges

- **Unsigned Values**
  - *UMin* = 0
    000…0
  - *UMax* = $2^w - 1$
    111…1

- **Two's Complement Values**
  - *TMin* = -2^{w-1}
    100…0
  - *TMax* = $2^{w-1} - 1$
    011…1
  - Minus 1
    111…1

Values for *W* = 16

	Decimal	Hex	Binary
UMax	65535	FF FF	11111111 11111111
TMax	32767	7F FF	01111111 11111111
TMin	-32768	80 00	10000000 00000000
-1	-1	FF FF	11111111 11111111
0	0	00 00	00000000 00000000

Values for Different Word Sizes

	W			
	8	**16**	**32**	**64**
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

- **Observations**
  - $|TMin| = TMax + 1$
    - Asymmetric range
  - $UMax = 2 * TMax + 1$
  - Question: abs(TMin)?

- **C Programming**
  - #include <limits.h>
  - Declares constants, e.g.,
    - ULONG_MAX
    - LONG_MAX
    - LONG_MIN
  - Values platform specific

Unsigned & Signed Numeric Values

X	B2U(X)	B2T(X)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

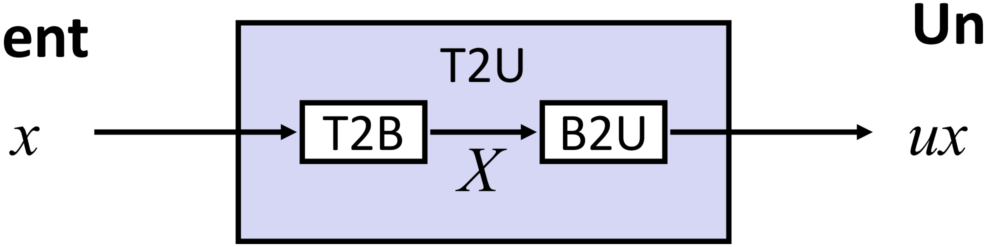
- **Equivalence**
  - Same encodings for nonnegative values
- **Uniqueness**
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding
- **⇒ Can Invert Mappings**
  - U2B(x) = B2U^{-1}(x)
    - Bit pattern for unsigned integer
  - T2B(x) = B2T^{-1}(x)
    - Bit pattern for two's comp integer

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - **Conversion, casting**
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings

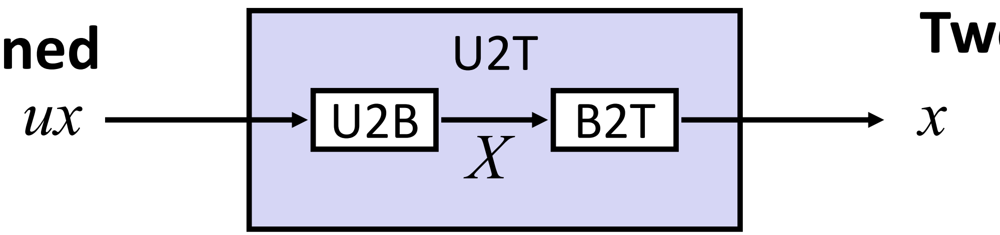
Mapping Between Signed & Unsigned

Two's Complement **Unsigned**



Maintain Same Bit Pattern

Unsigned **Two's Complement**



Maintain Same Bit Pattern

- **Mappings between unsigned and two's complement numbers:**
  Keep bit representations and reinterpret

Mapping Signed ↔ Unsigned

Bits	Signed	Unsigned
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	-8	8
1001	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

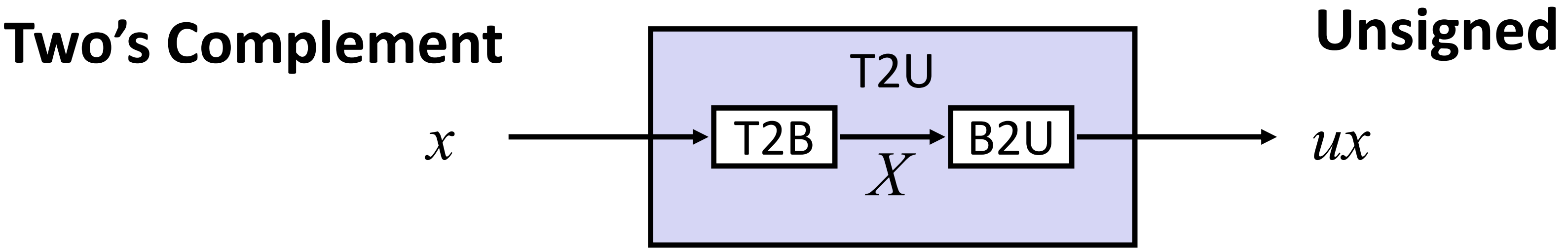
Signed $\xrightarrow{\text{T2U}}$ Unsigned

Signed $\xleftarrow{\text{U2T}}$ Unsigned

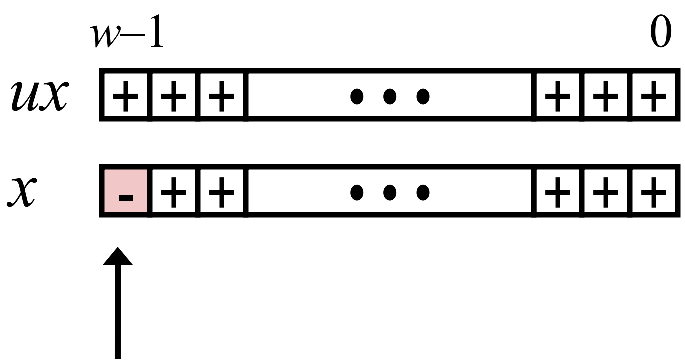
Mapping Signed ↔ Unsigned

Bits	Signed		Unsigned
0000	0		0
0001	1		1
0010	2		2
0011	3		3
0100	4	=	4
0101	5		5
0110	6		6
0111	7		7
1000	-8		8
1001	-7		9
1010	-6		10
1011	-5	+/- 16	11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

Relation between Signed & Unsigned



Maintain Same Bit Pattern



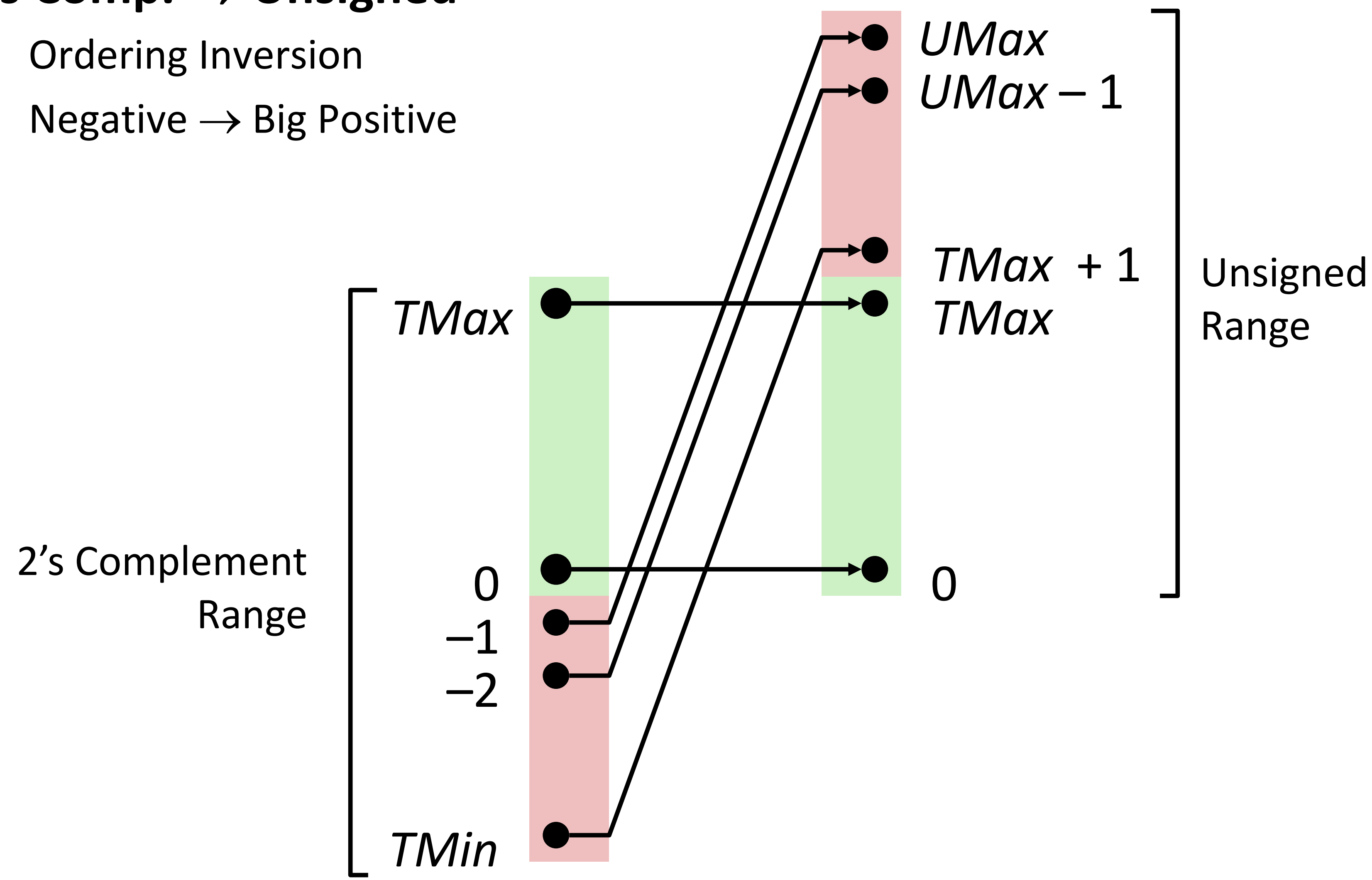
Large negative weight

becomes

Large positive weight

Conversion Visualized

- **2's Comp. → Unsigned**
  - Ordering Inversion
  - Negative → Big Positive



Signed vs. Unsigned in C

- **Constants**
  - By default are considered to be signed integers
  - Unsigned if have “U” as suffix

    `0U, 4294967259U`

- **Casting**
  - Explicit casting between signed & unsigned same as U2T and T2U

```
int tx, ty;
unsigned ux, uy;
tx = (int) ux;
uy = (unsigned) ty;
```

  - Implicit casting also occurs via assignments and procedure calls

```
tx = ux;                        int fun(unsigned u);
uy = ty;                        uy = fun(tx);
```

Casting Surprises

- **Expression Evaluation**
  - If there is a mix of unsigned and signed in single expression, ***signed values implicitly cast to unsigned***
  - Including comparison operations **<, >, ==, <=, >=**
  - Examples for $W = 32$: **TMIN = -2,147,483,648 ,** **TMAX = 2,147,483,647**

Constant$_1$	**Constant$_2$**	**Relation**	**Evaluation**
0	0U	**==**	**unsigned**
-1	0	**<**	**signed**
-1	0U	**>**	**unsigned**
2147483647	-2147483647-1	**>**	**signed**
2147483647U	-2147483647-1	**<**	**unsigned**
-1	-2	**>**	**signed**
(unsigned)-1	-2	**>**	**unsigned**
2147483647	2147483648U	**<**	**unsigned**
2147483647	(int) 2147483648U	**>**	**signed**

Summary
Casting Signed ↔ Unsigned: Basic Rules

- Bit pattern is maintained
- But reinterpreted
- Can have unexpected effects: adding or subtracting 2^w

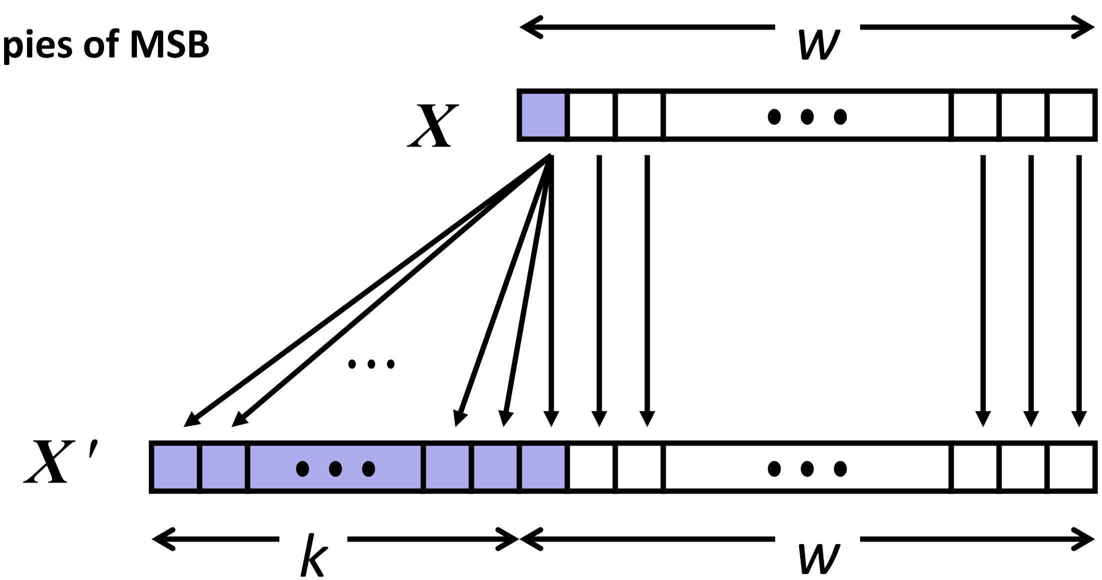
- Expression containing signed and unsigned int
  - `int` is cast to `unsigned`!!

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - **Expanding, truncating**
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings

Sign Extension

- **Task:**
  - Given *w*-bit signed integer *x*
  - Convert it to *w*+*k*-bit integer with same value
- **Rule:**
  - Make *k* copies of sign bit:
  - $X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \ldots, x_0$



Sign Extension: Simple Example

Positive number

	-16	8	4	2	1
10 =	0	1	0	1	0

	-32	16	8	4	2	1
10 =	0	0	1	0	1	0

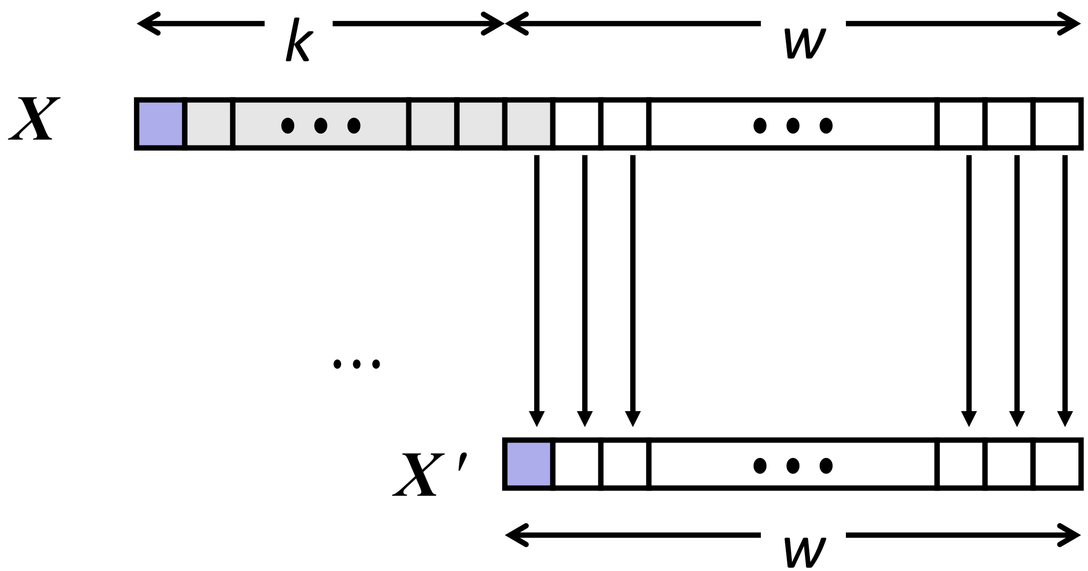
Negative number

	-16	8	4	2	1
-10 =	1	0	1	1	0

	-32	16	8	4	2	1
-10 =	1	1	0	1	1	0

Truncation

- **Task:**
  - Given k+w-bit signed or unsigned integer X
  - Convert it to w-bit integer X' with same value for "small enough" X
- **Rule:**
  - Drop top k bits:
  - $X' = x_{w-1}, x_{w-2}, \ldots, x_0$



Truncation: Simple Example

No sign change

	-16	8	4	2	1
2 =	0	0	0	1	0

	-8	4	2	1
2 =	0	0	1	0

`2 mod 16 = 2`

	-16	8	4	2	1
-6 =	1	1	0	1	0

	-8	4	2	1
-6 =	1	0	1	0

`-6 mod 16 = 26U mod 16 = 10U = -6`

Sign change

	-16	8	4	2	1
10 =	0	1	0	1	0

	-8	4	2	1
-6 =	1	0	1	0

`10 mod 16 = 10U mod 16 = 10U = -6`

	-16	8	4	2	1
-10 =	1	0	1	1	0

	-8	4	2	1
6 =	0	1	1	0

`-10 mod 16 = 22U mod 16 = 6U = 6`

Summary:
Expanding, Truncating: Basic Rules

- **Expanding (e.g., short int to int)**
  - Unsigned: zeros added
  - Signed: sign extension
  - Both yield expected result

- **Truncating (e.g., unsigned to unsigned short)**
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
  - For small (in magnitude) numbers yields expected behavior

Summary of Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
- Representations in memory, pointers, strings
- Summary

Today: Bits, Bytes, and Integers

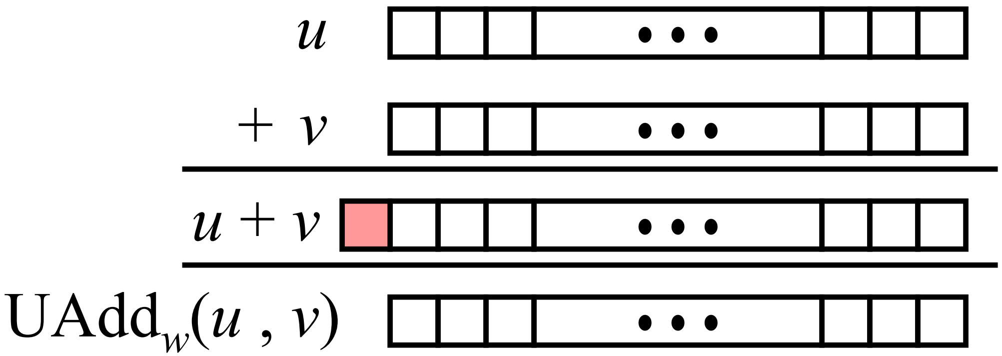
- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - **Addition, negation, multiplication, shifting**
- Representations in memory, pointers, strings
- Summary

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



- **Standard Addition Function**
  - Ignores carry output
- **Implements Modular Arithmetic**

$$s \quad = \quad \mathrm{UAdd}_w(u\,,v) \quad = \quad u + v \mod 2^w$$

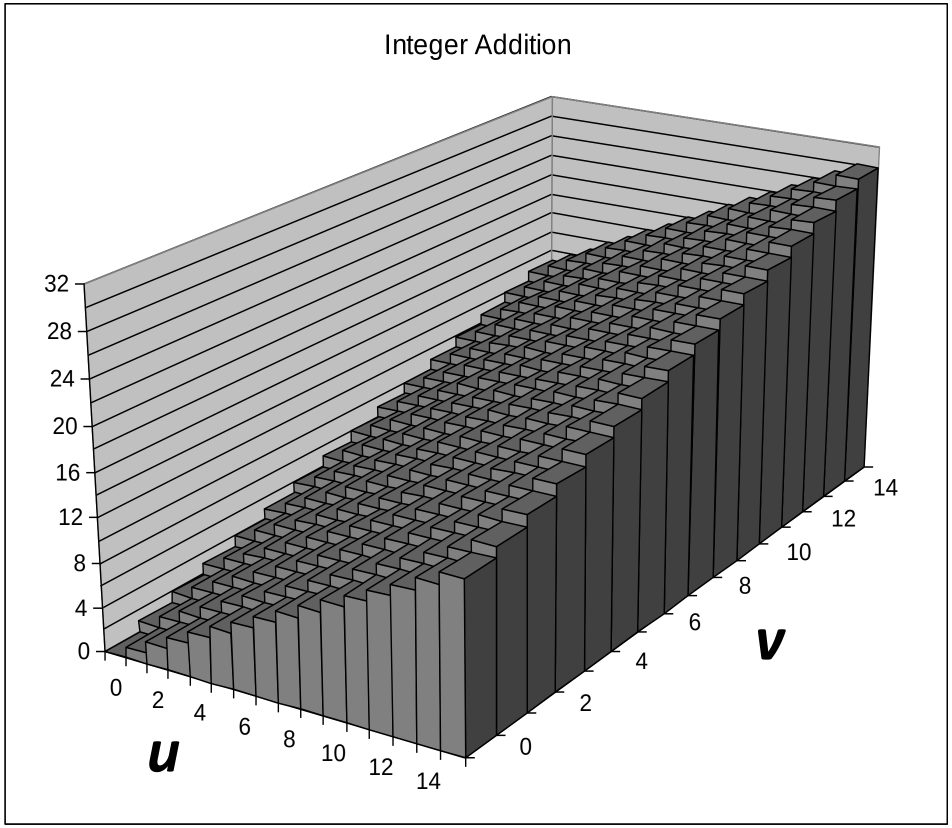
```
unsigned char     1110 1001      E9       223
                + 1101 0101    + D5     + 213
                 1 1011 1110    1BE       446
                  1011 1110      BE       190
```

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

Visualizing (Mathematical) Integer Addition

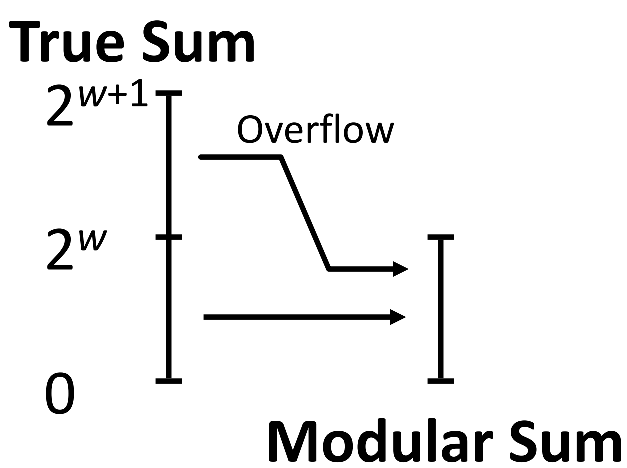
- **Integer Addition**
  - 4-bit integers *u*, *v*
  - Compute true sum $Add_4(u, v)$
  - Values increase linearly with *u* and *v*
  - Forms planar surface

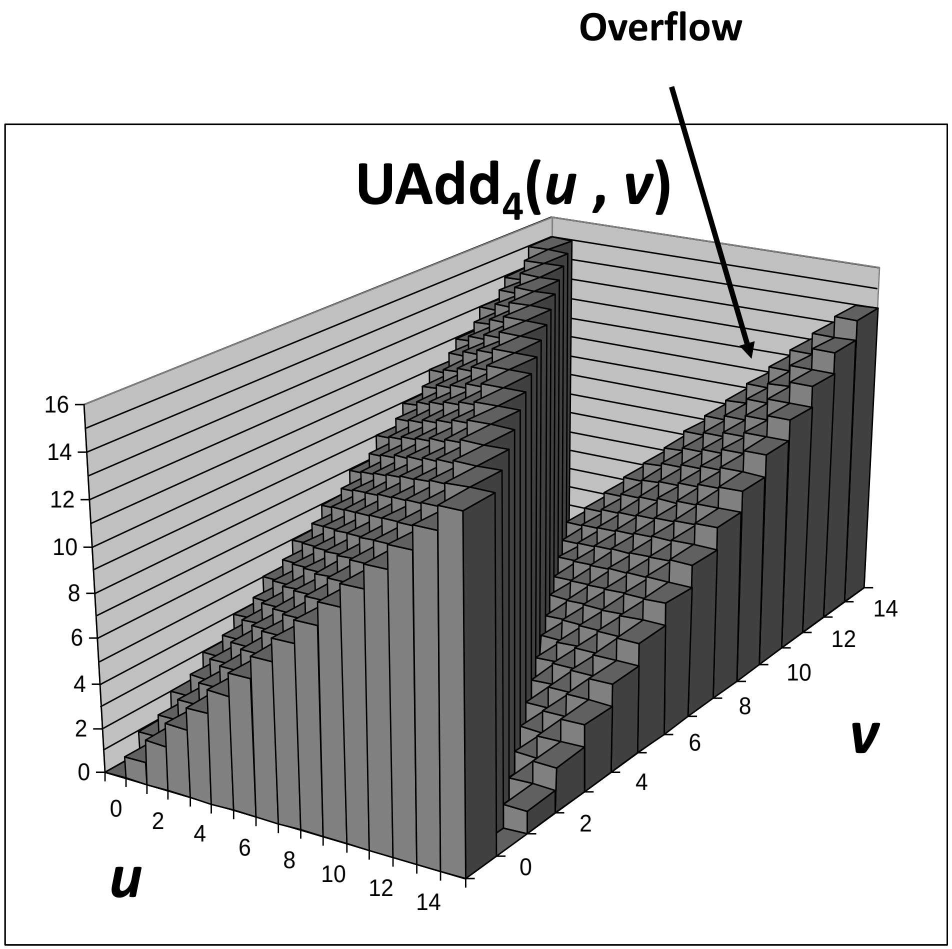
$Add_4(u, v)$



Visualizing Unsigned Addition

- **Wraps Around**
  - If true sum ≥ 2^w
  - At most once



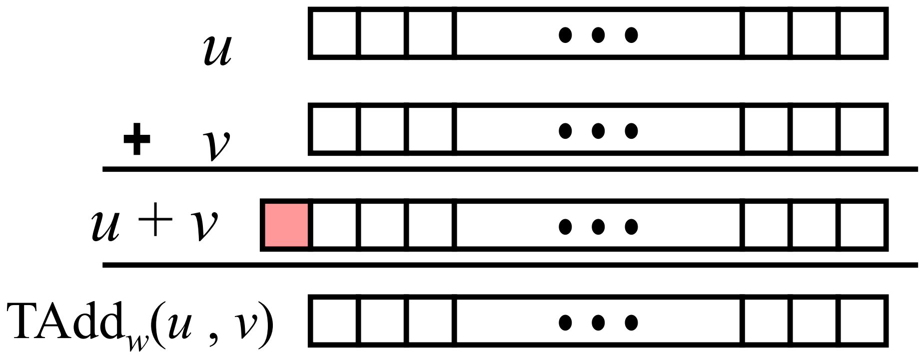


Two's Complement Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



- **TAdd and UAdd have Identical Bit-Level Behavior**
  - Signed vs. unsigned addition in C:

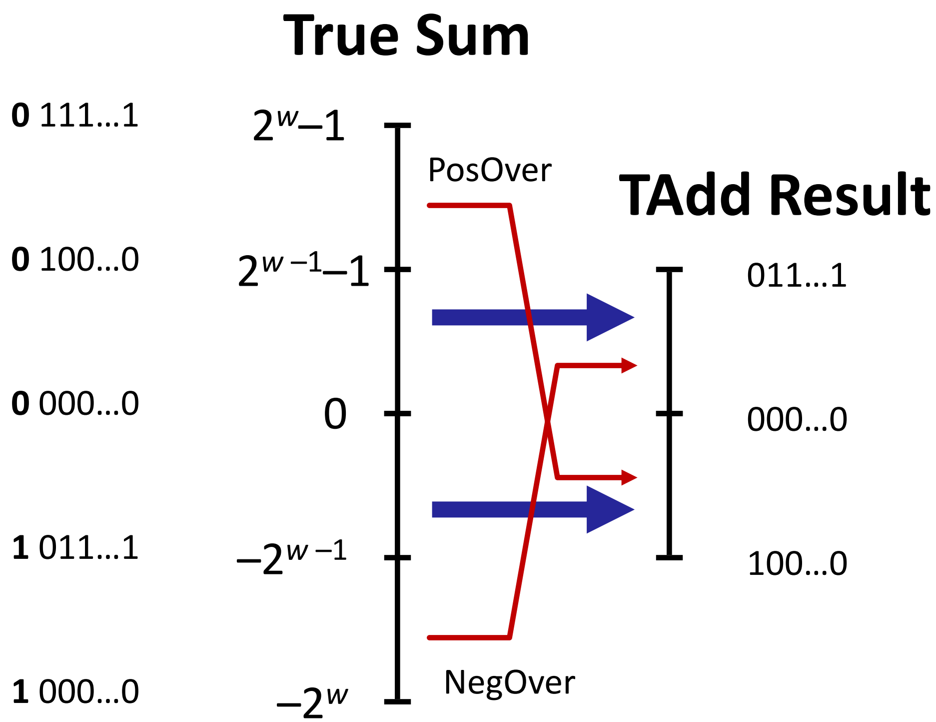
```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

  - Will give `s == t`

1110 1001	E9	-23
+ 1101 0101	+ D5	+ -43
1 1011 1110	1BE	-66
1011 1110	BE	-66

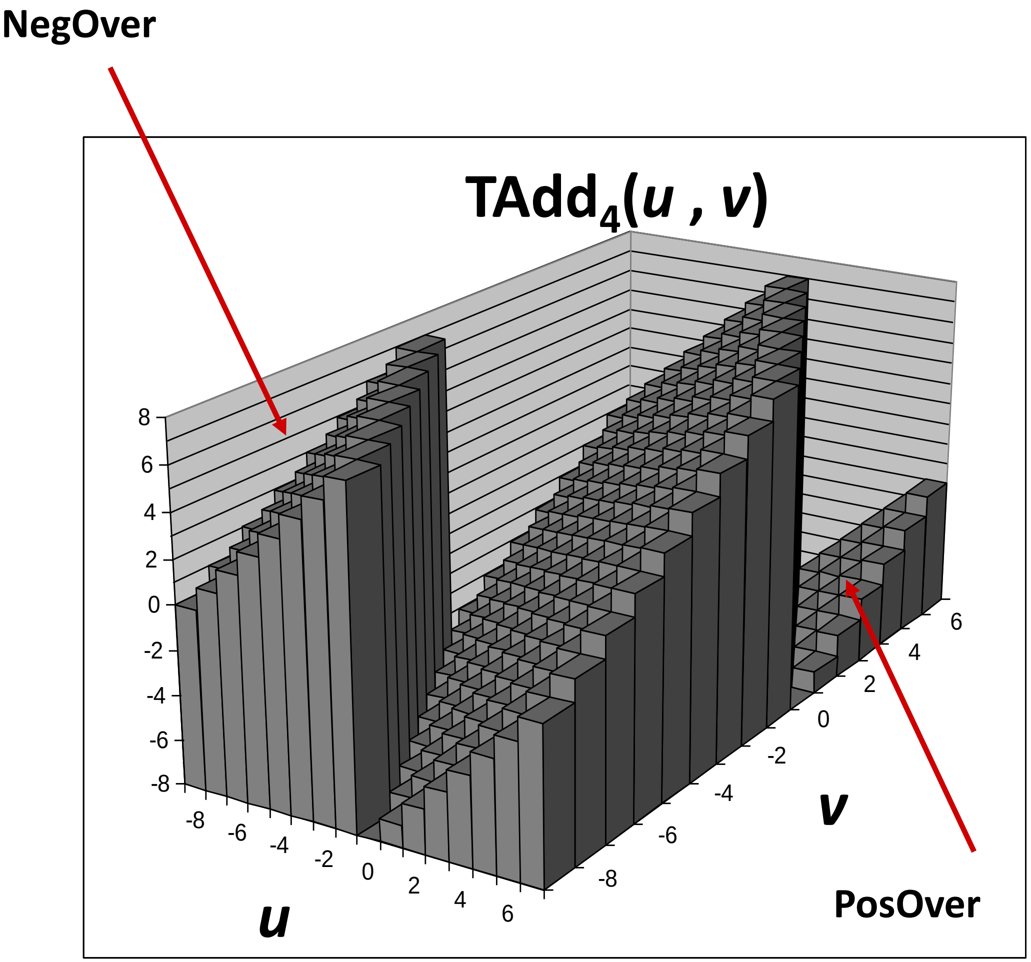
TAdd Overflow

- **Functionality**
  - True sum requires w+1 bits
  - Drop off MSB
  - Treat remaining bits as 2's comp. integer



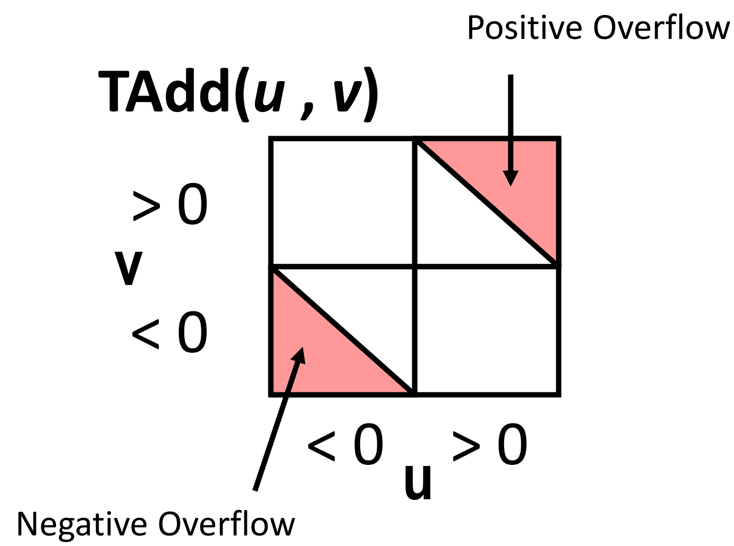
Visualizing 2's Complement Addition

- **Values**
  - 4-bit two's comp.
  - Range from -8 to +7
- **Wraps Around**
  - If sum $\geq 2^{w-1}$
    - Becomes negative
    - At most once
  - If sum $< -2^{w-1}$
    - Becomes positive
    - At most once



Characterizing TAdd

- **Functionality**
  - True sum requires *w*+1 bits
  - Drop off MSB
  - Treat remaining bits as 2's comp. integer



$$TAdd_w(u,v) \quad = \begin{cases} u+v+2^w & u+v < TMin_w \quad \textbf{(NegOver)} \\ u+v & TMin_w \leq u+v \leq TMax_w \\ u+v-2^w & TMax_w < u+v \quad \textbf{(PosOver)} \end{cases}$$

Multiplication

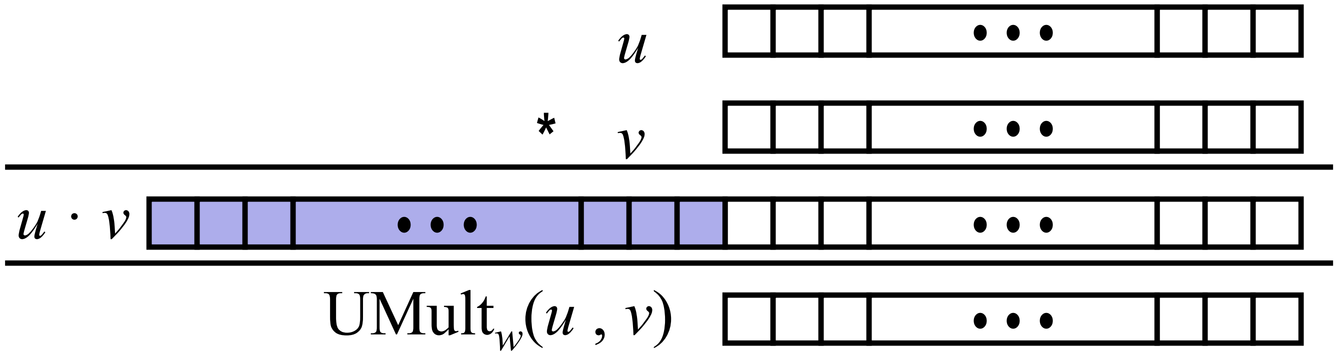
- **Goal: Computing Product of *w*-bit numbers *x*, *y***
  - Either signed or unsigned
- **But, exact results can be bigger than *w* bits**
  - Unsigned: up to $2w$ bits
    - Result range: $0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Two's complement min (negative): Up to $2w$-1 bits
    - Result range: $x * y \ge (-2^{w-1})*(2^{w-1}-1) = -2^{2w-2} + 2^{w-1}$
  - Two's complement max (positive): Up to $2w$ bits, but only for $(TMin_w)^2$
    - Result range: $x * y \le (-2^{w-1})^2 = 2^{2w-2}$
- **So, maintaining exact results…**
  - would need to keep expanding word size with each product computed
  - is done in software, if needed
    - e.g., by "arbitrary precision" arithmetic packages

Unsigned Multiplication in C

Operands: *w* bits

True Product: 2**w* bits

Discard *w* bits: *w* bits



- **Standard Multiplication Function**
  - Ignores high order *w* bits
- **Implements Modular Arithmetic**

$\mathrm{UMult}_w(u, v) = u \cdot v \bmod 2^w$

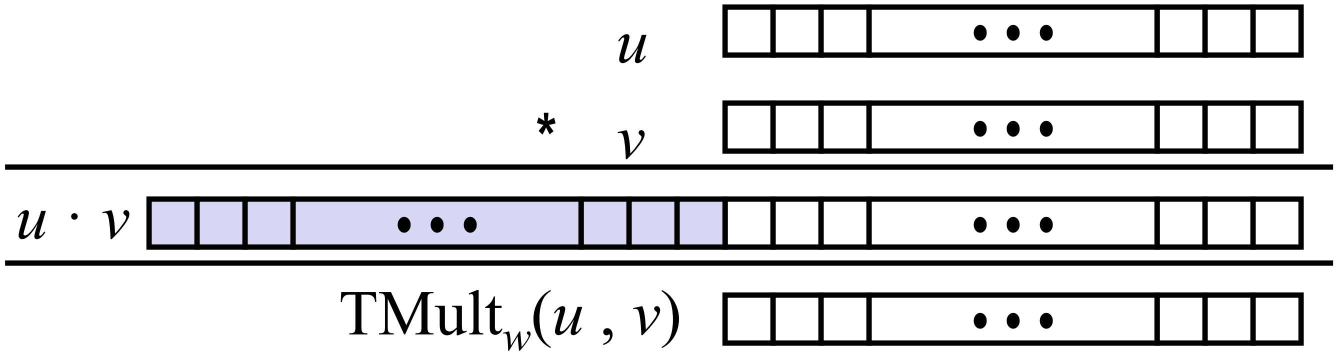
	1110 1001	E9	233
*	1101 0101	* D5	* 213
	1100 0001 1101 1101	C1DD	49629
	1101 1101	DD	221

Signed Multiplication in C

Operands: *w* bits

True Product: 2**w* bits

Discard *w* bits: *w* bits



- **Standard Multiplication Function**
  - Ignores high order *w* bits
  - Some of which are different for signed vs. unsigned multiplication
  - Lower bits are the same

```
                1110 1001      E9          -23
*               1101 0101    * D5     *    -43
 0000 0011 1101 1101         03DD          989
                1101 1101      DD          -35
```

Power-of-2 Multiply with Shift

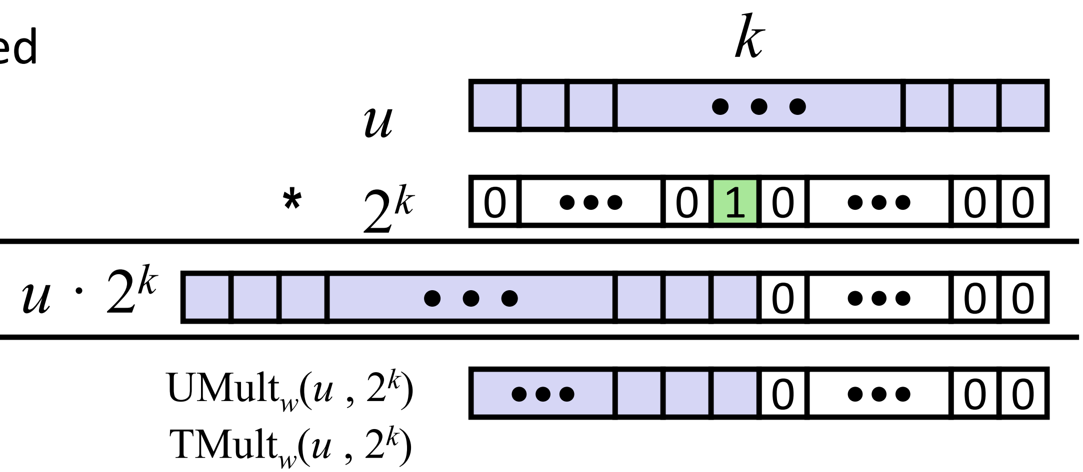
Operation

- `u << k` gives `u * 2`^{k}
- Both signed and unsigned

Operands: *w* bits

True Product: *w*+*k* bits

Discard *k* bits: *w* bits

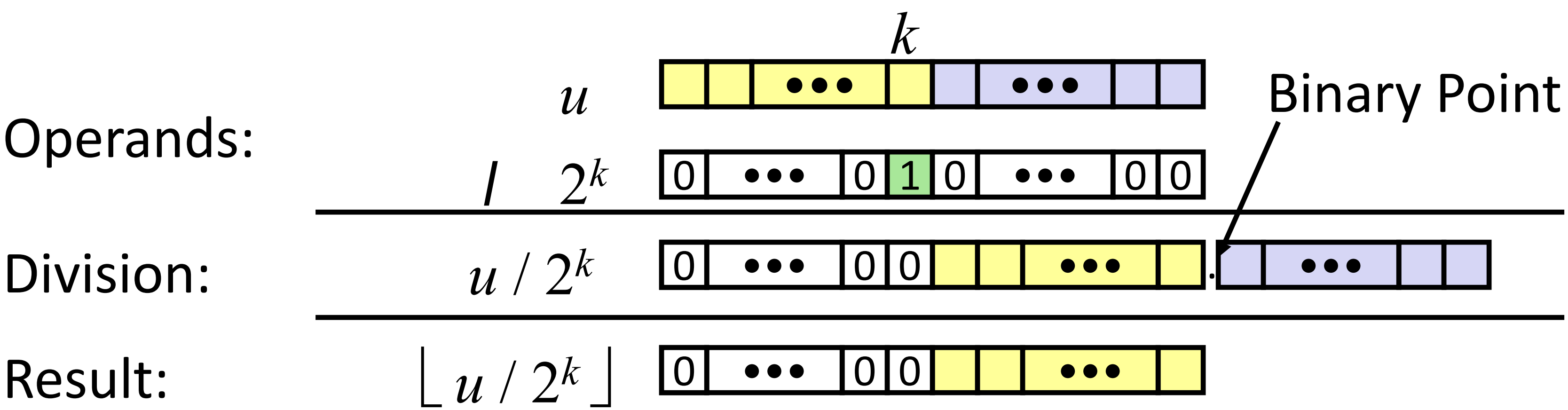


Examples

- `u << 3 == u * 8`
- `(u << 5) – (u << 3) == u * 24`
- Most machines shift and add faster than multiply
  - Compiler generates this code automatically

Unsigned Power-of-2 Divide with Shift

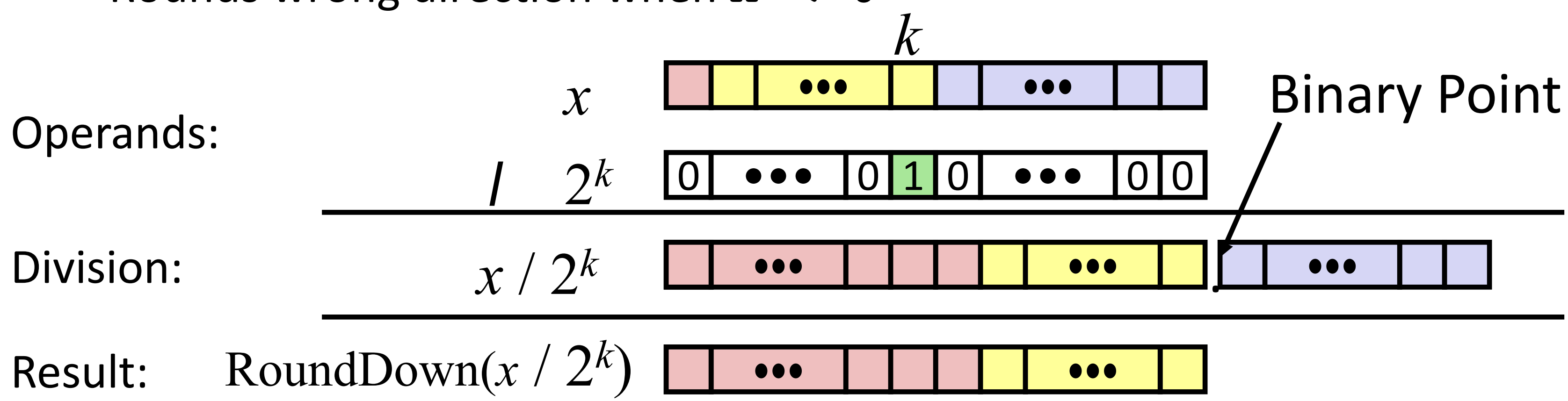
- **Quotient of Unsigned by Power of 2**
  - `u >> k` gives $\lfloor$ `u / 2`^{k} $\rfloor$
  - Uses logical shift



	Division	Computed	Hex	Binary
`x`	15213	15213	`3B 6D`	`00111011 01101101`
`x >> 1`	7606.5	7606	`1D B6`	`00011101 10110110`
`x >> 4`	950.8125	950	`03 B6`	`00000011 10110110`
`x >> 8`	59.4257813	59	`00 3B`	`00000000 00111011`

Signed Power-of-2 Divide with Shift

- **Quotient of Signed by Power of 2**
  - `x >> k` gives $\lfloor$ `x / 2`^{k} $\rfloor$
  - Uses arithmetic shift
  - Rounds wrong direction when `x < 0`



	Division	Computed	Hex	Binary
`x`	-15213	-15213	`C4 93`	`11000100 10010011`
`x >> 1`	-7606.5	-7607	`E2 49`	`11100010 01001001`
`x >> 4`	-950.8125	-951	`FC 49`	`11111100 01001001`
`x >> 8`	-59.4257813	-60	`FF C4`	`11111111 11000100`

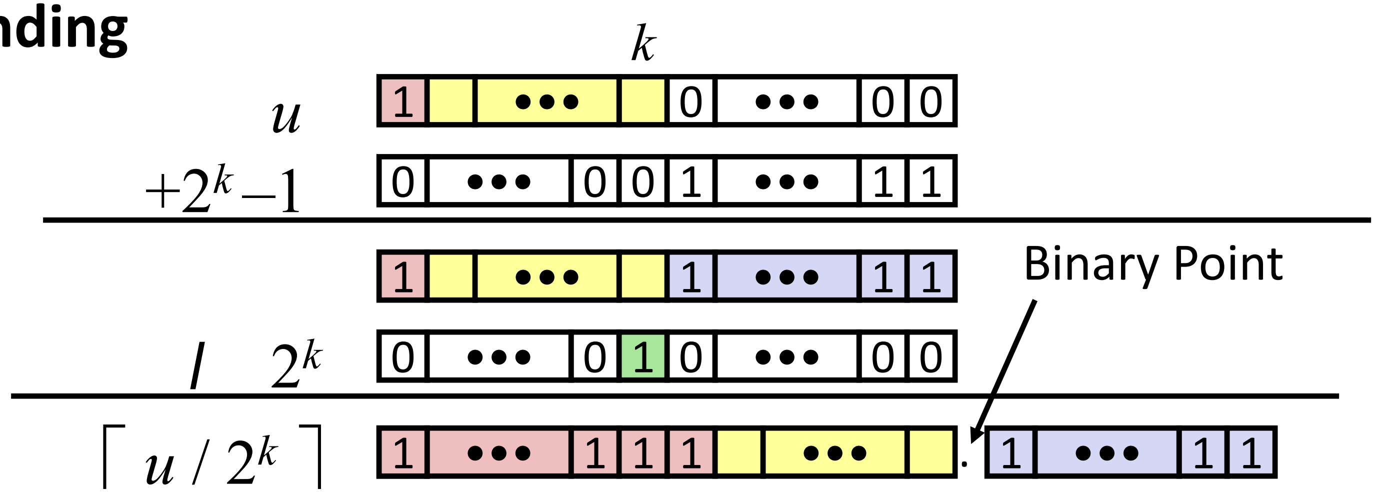
Correct Power-of-2 Divide

- **Quotient of Negative Number by Power of 2**
  - Want $\lceil$ **`x / 2`**^{k} $\rceil$ (Round Toward 0)
  - Compute as $\lfloor$ **`(x+2`**^{k}**`-1)/ 2`**^{k} $\rfloor$
    - In C: **`(x + (1<<k)-1) >> k`**
    - Biases dividend toward 0

Case 1: No rounding

Dividend:

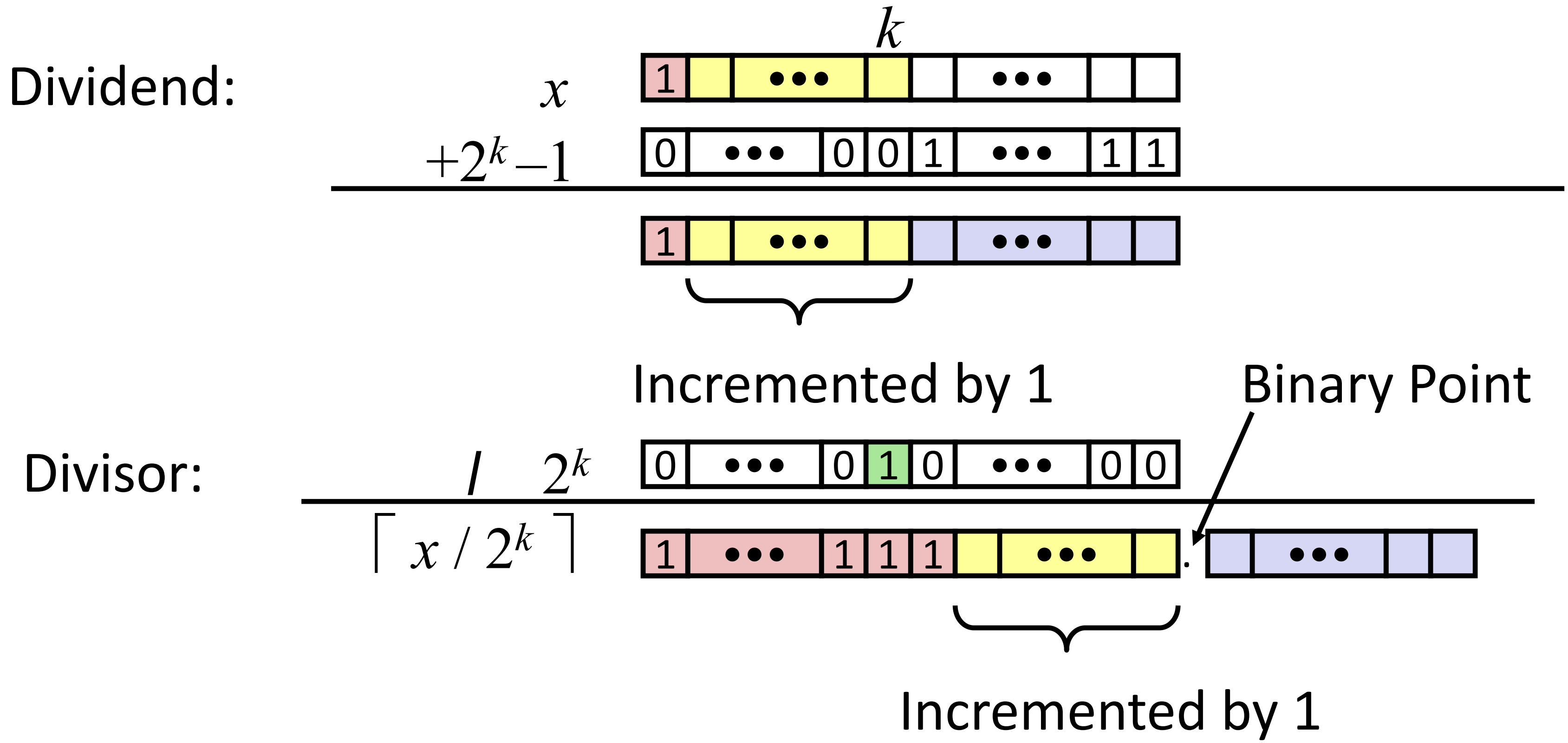
Divisor:



Biasing has no effect

Correct Power-of-2 Divide (Cont.)

Case 2: Rounding



Biasing adds 1 to final result

Today: Bits, Bytes, and Integers

- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - **Summary**
- Representations in memory, pointers, strings

Arithmetic: Basic Rules

- **Addition:**
  - Unsigned/signed: Normal addition followed by truncate, same operation on bit level
  - Unsigned: addition mod 2^w
    - Mathematical addition + possible subtraction of 2^w
  - Signed: modified addition mod 2^w (result in proper range)
    - Mathematical addition + possible addition or subtraction of 2^w

- **Multiplication:**
  - Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
  - Unsigned: multiplication mod 2^w
  - Signed: modified multiplication mod 2^w (result in proper range)