Jin-Soo Kim
(jinsoo.kim@snu.ac.kr)

Systems Software &
Architecture Lab.

Seoul National University

Spring 2019

Integers

Computer Systems











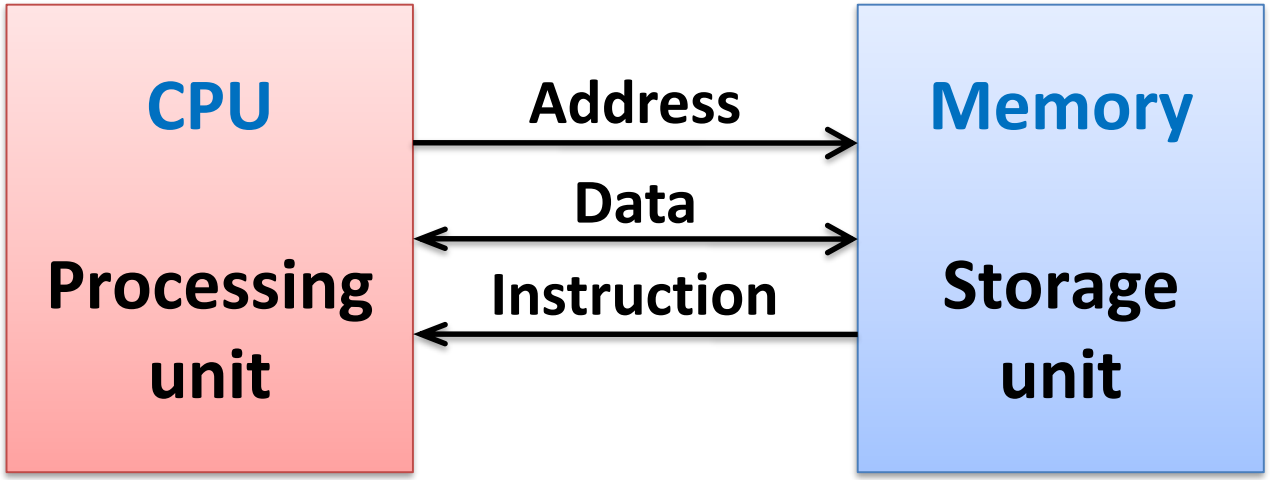





A computer is a [ ] machine.

Von Neumann Architecture

- By John von Neumann, 1945

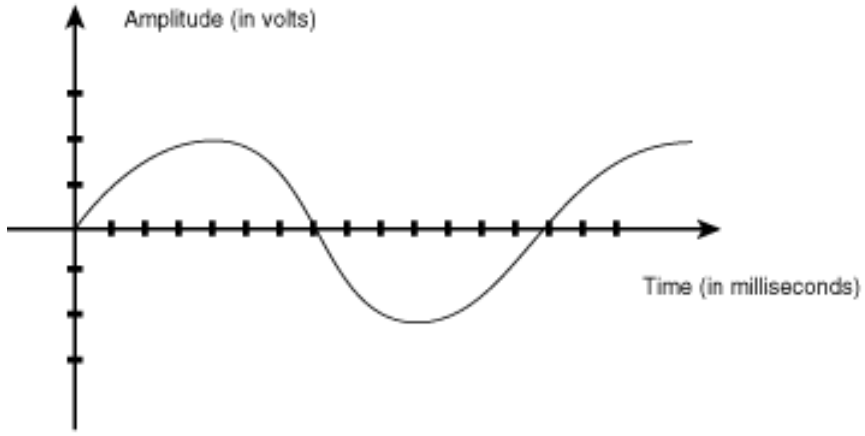


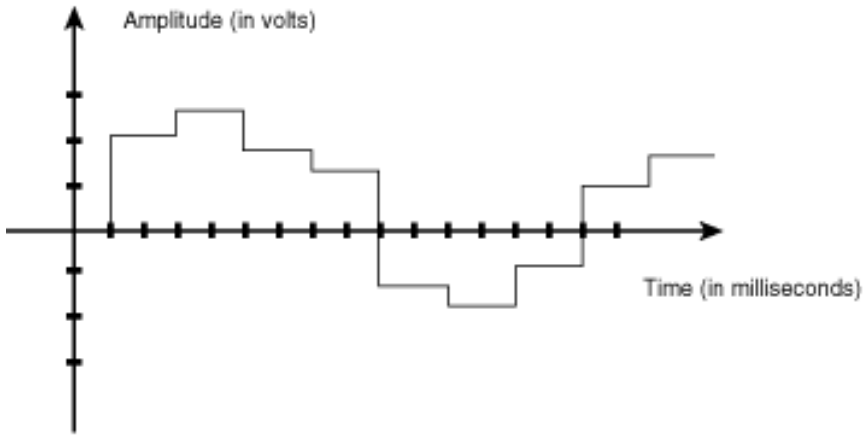
Data movement
Arithmetic & logical ops
Control transfer

Byte addressable array
Code + data (user program, OS)
Stack to support procedures

The Advent of the Digital Age

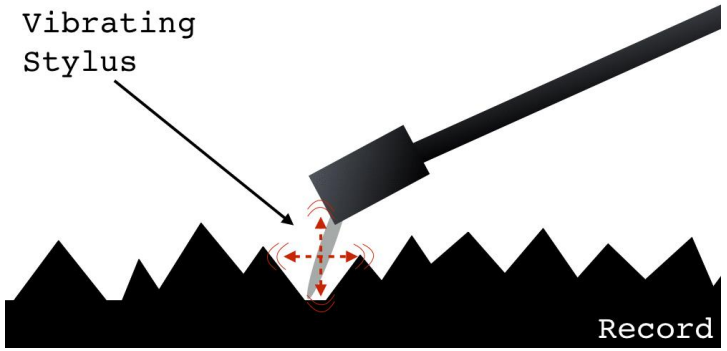
- Analog vs. digital?

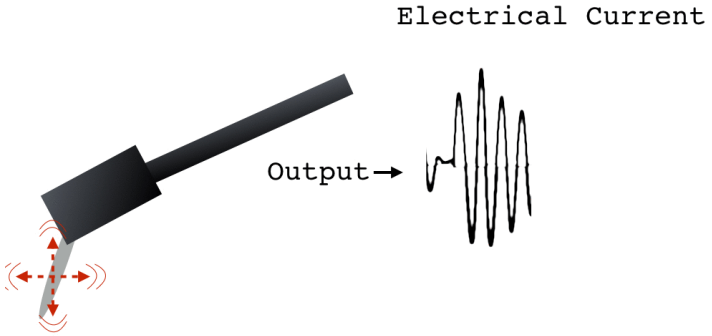


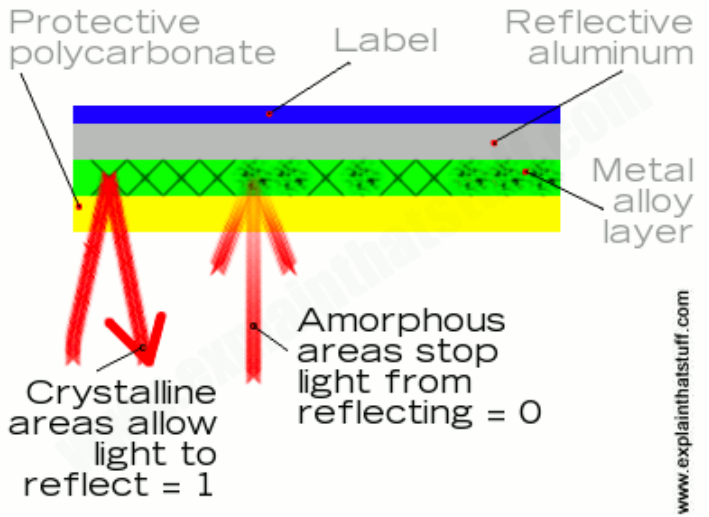


- Compact Disc (CD)
  - 44.1 KHz, 16-bit, 2-channel
- MP3
  - A digital audio encoding with lossy data compression

LP Record vs. Compact Disc

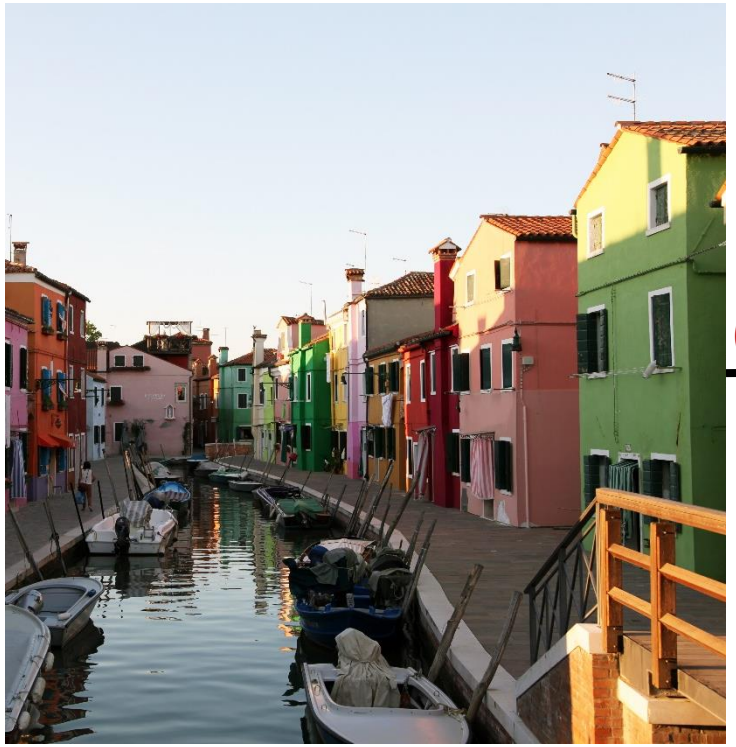


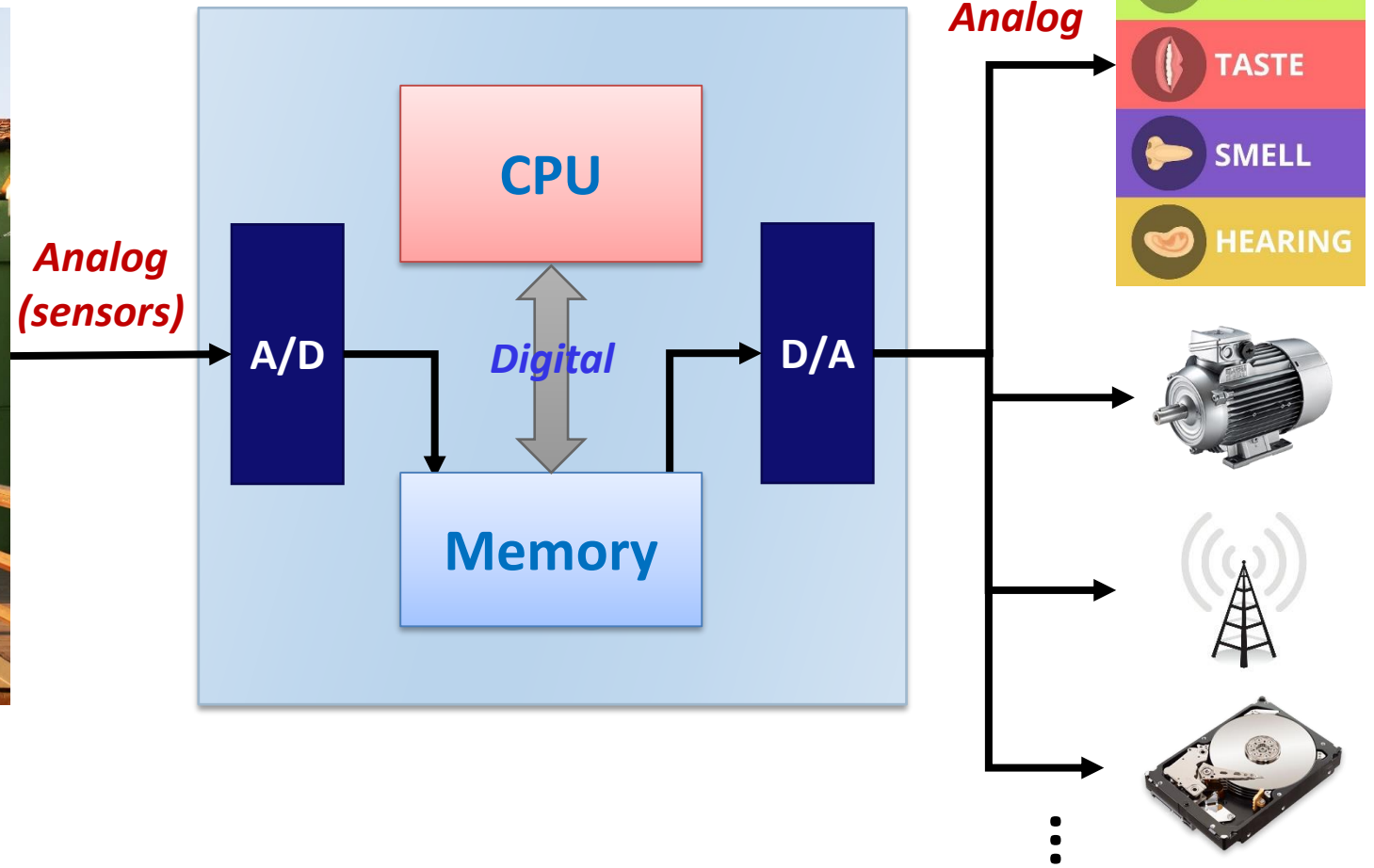




Source: http://www.soundsetal.com/blog-how-do-vinyl-records-work/
http://www.explainthatstuff.com/cdplayers.html

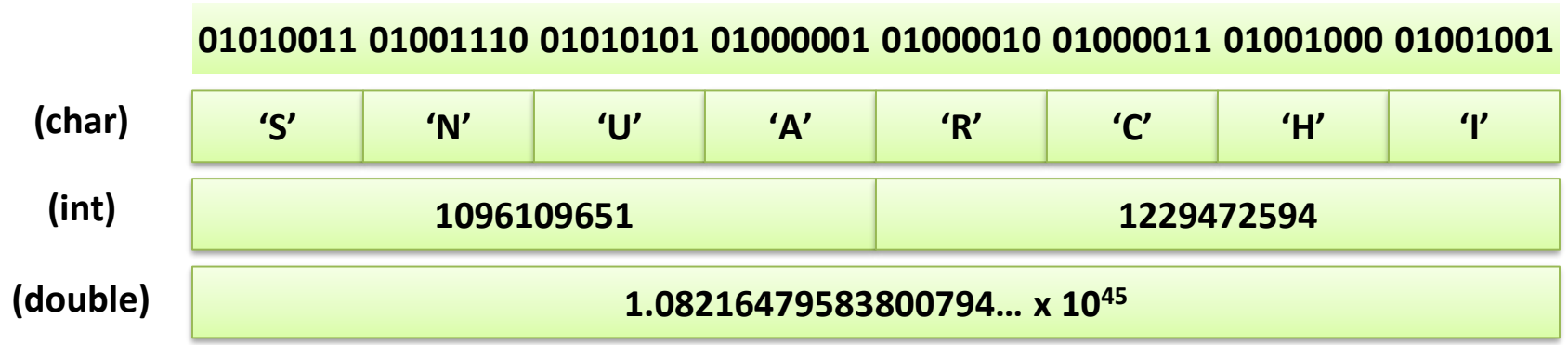
Digital Computer





Representing Information

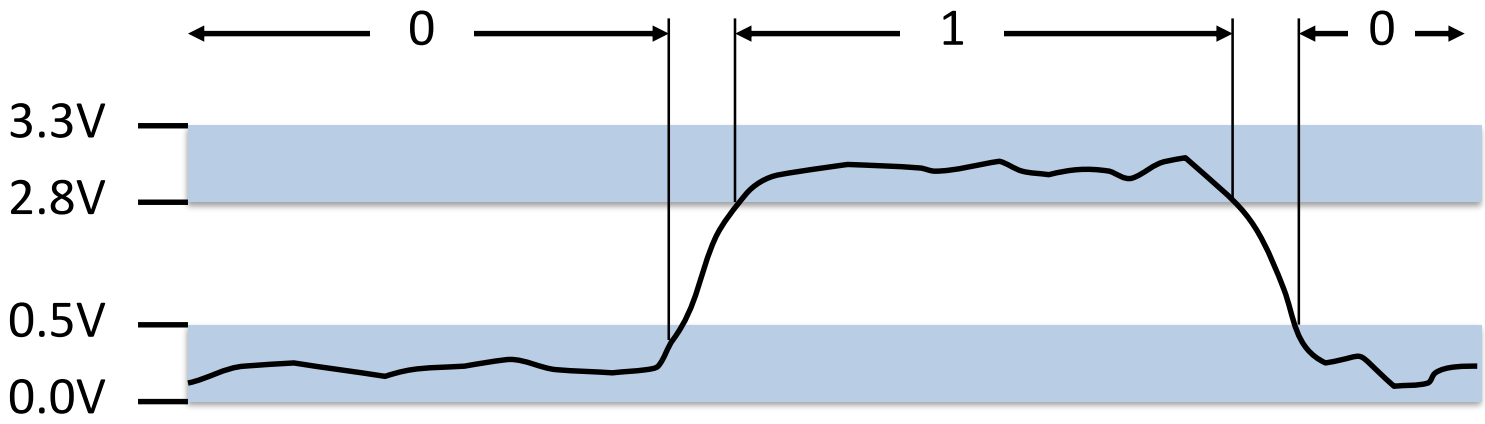
- Information = Bits + Context
  - Computers manipulate representations of things
  - Things are represented as binary digits
  - What can you represent with *N* bits?
    - 2^N things
    - Numbers, characters, pixels, positions, source code, executable files, machine instructions, …
    - Depends on what operations you do on them



Binary Representations

- Why not base 10 representation?
  - Easy to store with bistable elements
  - Straightforward implementation of arithmetic functions
  - Reliably transmitted on noisy and inaccurate wires

- Electronic implementation



Encoding Byte Values

- Binary: 00000000_2 to 11111111_2
- Octal: 000_8 to 377_8
  - An integer constant that begins with `0` is an octal number in C
- Decimal: 0_{10} to 255_{10}
  - First digit must not be `0` in C
- Hexadecimal: 00_{16} to FF_{16}
  - Base 16 number representation
  - Use characters '0' to '9' and 'A' to 'F'
  - Write $FA1D37B_{16}$ in C as **`0xFA1D37B`** or **`0xfa1d37b`**

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

Representing Integers

Unsigned Integers

- Encoding unsigned integers

$$B = [b_{w-1}, b_{w-2}, \ldots, b_0]$$

x = 0001 0000 0101 1110$_2$

$$D(B) = \sum_{i=0}^{w-1} b_i \cdot 2^i$$

D(x) = $2^{12} + 2^6 + 2^4 + 2^3 + 2^2 + 2^1$
= 4096 + 64 + 16 + 8 + 4 + 2
= 4190

- What is the range for unsigned values with *w* bits?

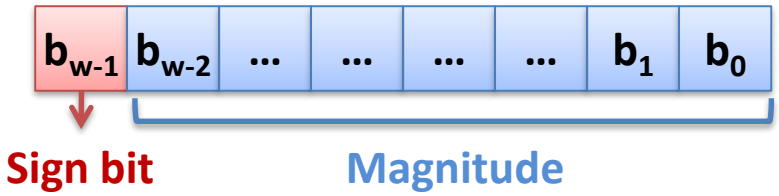
Signed Integers

- Encoding positive numbers
  - Same as unsigned numbers

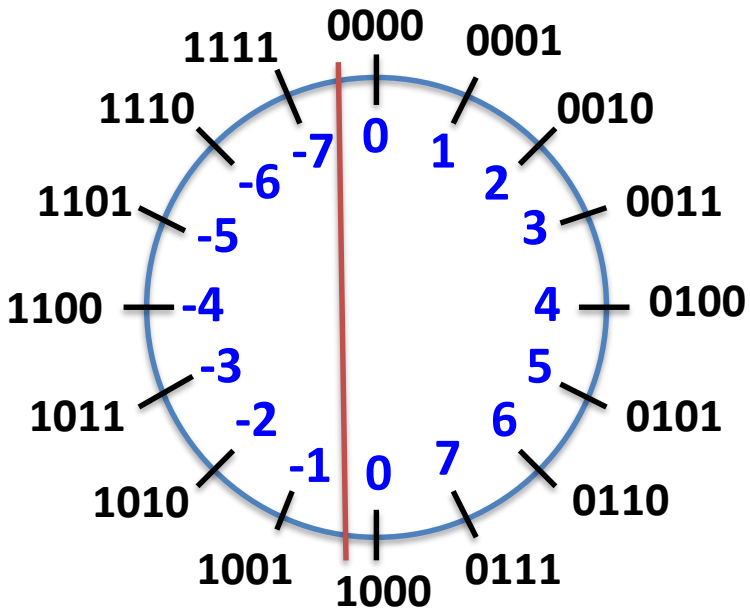
- Encoding negative numbers
  - Sign-magnitude representation
  - Ones' complement representation
  - Two's complement representation

Sign-magnitude Representation

- Two zeros
  - [000…00], [100..00]
- Used for floating-point numbers



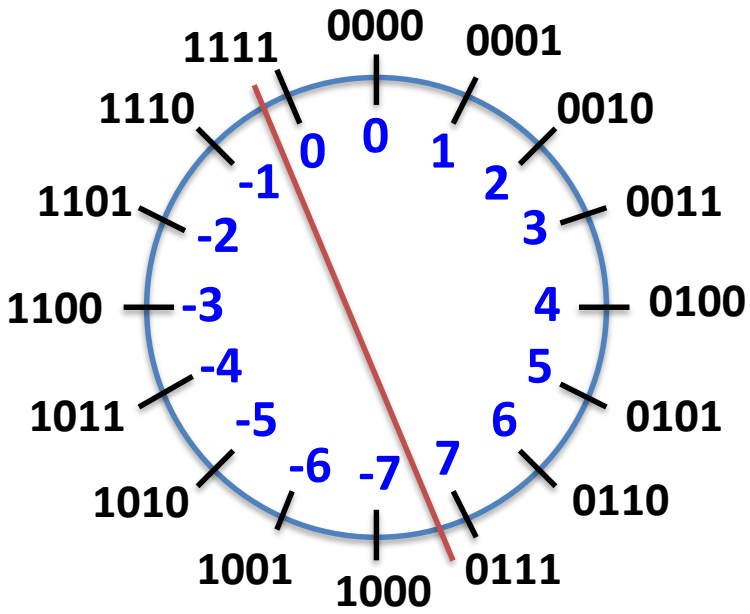
$$S(B) = (-1)^{b_{w-1}} \cdot \left(\sum_{i=0}^{w-2} b_i \cdot 2^i \right)$$



Ones' Complement Representation

- Easy to find –*n*
- Two zeros
  - [000..00], [111..11]
- No longer used

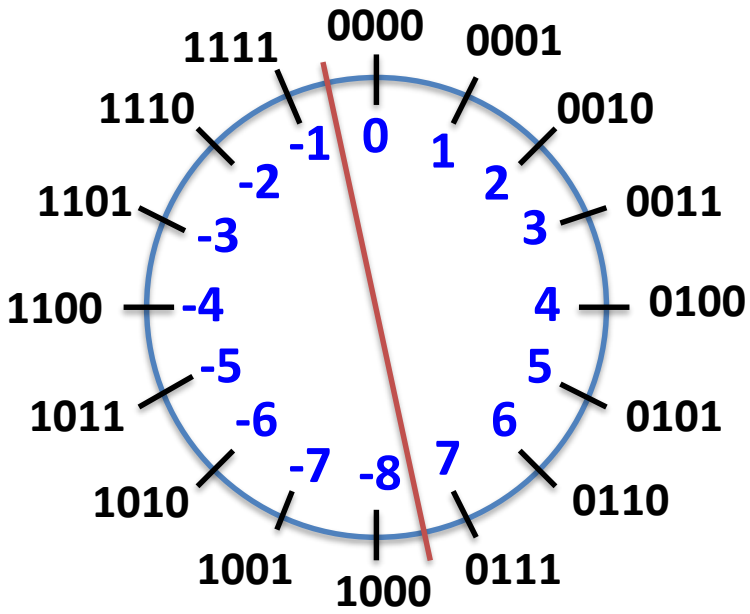
$$O(B) = -b_{w-1}(2^{w-1} - 1) + \left(\sum_{i=0}^{w-2} b_i \cdot 2^i \right)$$



Two's Complement Representation (1)

- Unique zero
- Easy for hardware
  - leading 0 ≥ 0
  - leading 1 < 0
- Used by almost all modern machines

$$O(B) = -b_{w-1} \cdot 2^{w-1} + \left(\sum_{i=0}^{w-2} b_i \cdot 2^i \right)$$



Two's Complement Representation (2)

- Following holds for two's complement

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

- Complement
  - Observation: $\sim x + x == 1111...11_2 == -1$

- Increment

$\sim x + x == -1$

$\sim x + x + (-x + 1) == -1 + (-x + 1)$

$\sim x + 1 == -x$

Numeric Ranges

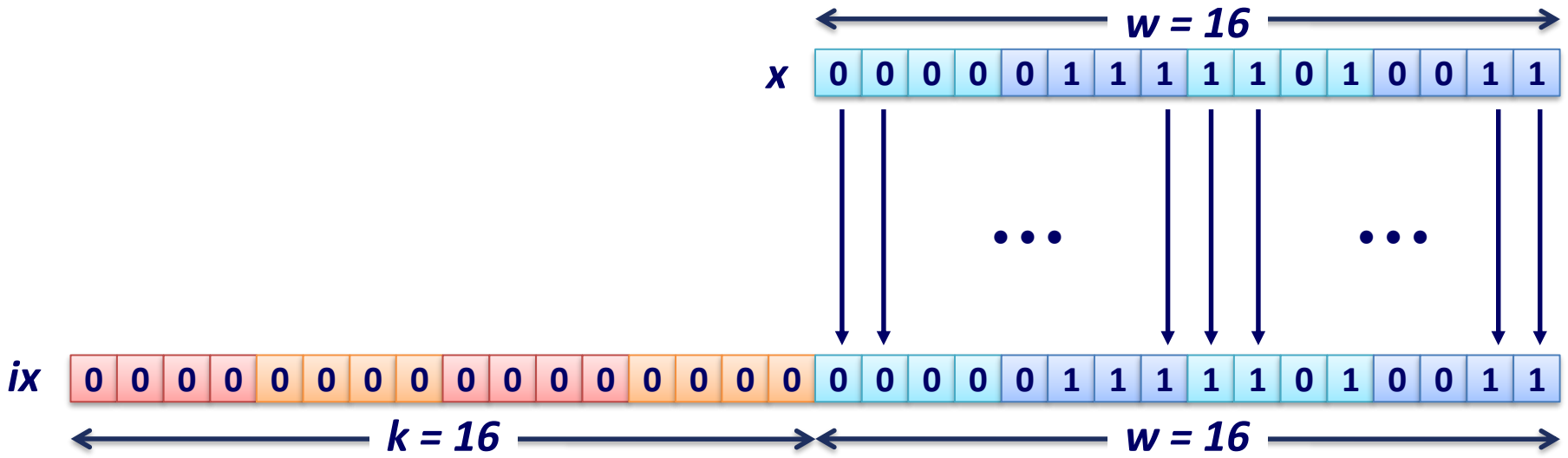
- Unsigned values
  - UMin = 0 [000…00]
  - UMax = $2^w - 1$ [111…11]
- Two's complement values
  - TMin = -2^{w-1} [100…00]
  - TMax = $2^{w-1} - 1$ [011…11]

	w = 8	w = 16	w = 32	w = 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

Type Conversion (1)

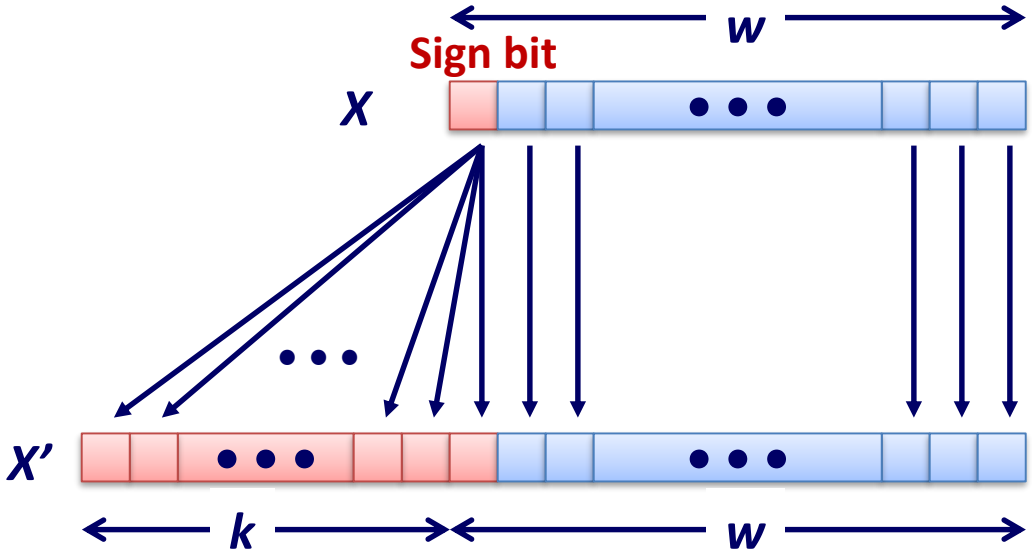
- Unsigned: ***w*** bits → ***w+k*** bits
  - Zero extension: just fill ***k*** bits with 0's

```
unsigned short  x  = 2003;
unsigned        ix = (unsigned) x;
```



Type Conversion (2)

- Signed: ***w*** bits → ***w+k*** bits
  - Given ***w***-bit signed integer ***x***
  - Convert it to ***w+k***-bit integer with same value
- Sign extension
  - Make k copies of sign bit



Type Conversion (3)

- Unsigned & Signed: ***w+k*** bits → ***w*** bits
  - Just truncate it to lower ***w*** bits
  - Equivalent to computing ***x*** mod 2^w

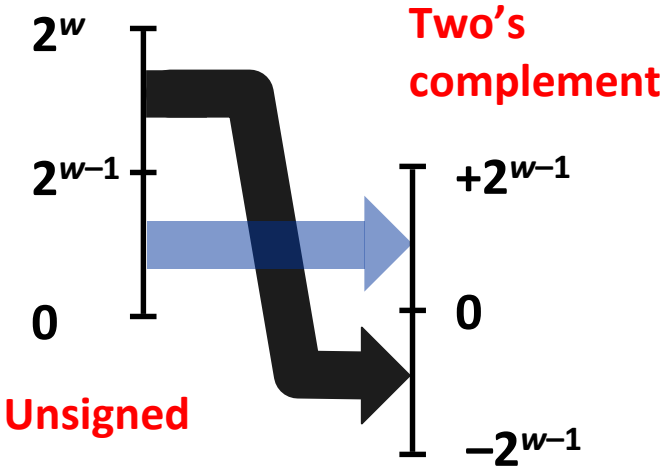
```
unsigned int      x  =  0xcafebabe;
unsigned short    ix =  (unsigned short) x;
int               y  =  0x2003beef;
short             iy =  (short) y;
```

	Decimal	Hex	Binary
x	3405691582	CA FE BA BE	11001010 11111110 10111010 10111110
ix	47806	BA BE	10111010 10111110
y	537116399	20 03 BE EF	00100000 00000011 10111110 11101111
iy	-16657	BE EF	10111110 11101111

Type Conversion (4)

- Unsigned → Signed
  - The same bit pattern is interpreted as a signed number

$$U2T_w(x) = \begin{cases} x, & x < 2^{w-1} \\ x - 2^w, & x \geq 2^{w-1} \end{cases}$$



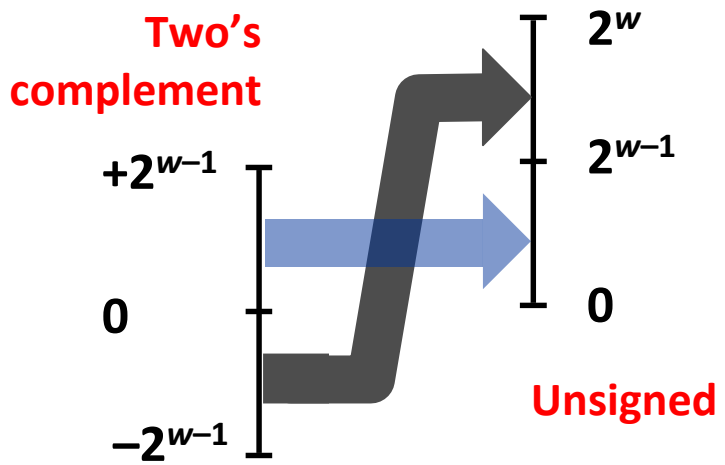

```
unsigned short x  = 2003;
short          ix = (short) x;
unsigned short y  = 0xbabe;
short          iy = (short) y;
```

	Decimal	Hex	Binary
x	2003	07 D3	00000111 11010011
ix	2003	07 D3	00000111 11010011
y	47806	BA BE	10111010 10111110
iy	-17730	BA BE	10111010 10111110

Type Conversion (5)

- Signed → Unsigned
  - Ordering inversion
  - Negative → Big positive

$$T2U_w(x) = \begin{cases} x + 2^w, & x < 0 \\ x, & x \geq 0 \end{cases}$$




```
short          x = 2003;
unsigned short ix = (unsigned short) x;
short          y = -2003;
unsigned short iy = (unsigned short) y;
```

	Decimal	Hex	Binary
x	2003	07 D3	00000111 11010011
ix	2003	07 D3	00000111 11010011
y	-2003	F8 2D	11111000 00101101
iy	63533	F8 2D	11111000 00101101

Type Casting in C

- Constants
  - By default, considered to be signed integers
  - Unsigned if they have "U" or "u" as suffix
    - e.g. `0U`, `12345U`, `0x1A2Bu`
- Type casting
  - Explicit casting

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

  - Implicit casting via
    - Assignments
    - Procedure calls

```
int f(unsigned);
tx = ux;
f(ty);
```

Expression Evaluation in C

- If mix unsigned and signed in single expression, signed values implicitly cast to unsigned
- Including comparison operations <, >, ==, <=, >=

Expression	Type	Evaluation
0 == 0U	unsigned	True
-1 < 0	signed	True
-1 < 0U	unsigned	?
-1 > -2	signed	?
(unsigned) -1 > -2	unsigned	?
2147483647 > -2147483647-1	signed	?
2147483647U > -2147483647-1	unsigned	?
2147483647 > (int) 2147483648U	signed	?

Example 1

```
#include <stdio.h>

int main ()
{
    unsigned  i;
    for  (i = 10;  i >= 0;  i--)
        printf  (“%u\n”,  i);
}
```

Example 2

```
#include <stdio.h>

#define DELTA   sizeof(int)

int main ()
{
    int  i;
    for  (i = 10;  i – DELTA >= 0;  i -= DELTA)
        printf  (“%d\n”,  i);
}
```

Example 3

```
#include <string.h>

int strlonger (char *s, char *t)
{
    return (strlen(s) – strlen(t)) > 0;
}
```

Lessons

- There are many tickly situations when you use unsigned integers – hard to debug
- Do not use just because numbers are nonnegative
- Use only when you need collections of bits with no numeric interpretation (“flags”)
- Few languages other than C support unsigned integers

Manipulating Integers

Bit-Level Operations in C

- Operations &, |, ~, ^ available in C
  - Apply to any "integral" data type
    - `(unsigned) long, int, short, char`
  - View arguments as bit vectors
  - Arguments applied bit-wise
- Examples (char data type)

`~0x41 → 0xBE`	$\sim 01000001_2 \rightarrow 10111110_2$
`~0x00 → 0xFF`	$\sim 00000000_2 \rightarrow 11111111_2$
`0x69 & 0x55 → 0x41`	$01101001_2 \;\&\; 01010101_2 \rightarrow 01000001_2$
`0x69 | 0x55 → 0x7D`	$01101001_2 \;\&\; 01010101_2 \rightarrow 01111101_2$
`0x69 ^ 0x55 → 0x3C`	$01101001_2 \;\&\; 01010101_2 \rightarrow 00111100_2$

Logic Operations in C

- &&, ||, !
  - 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
if (p && *p) …                  // avoids null pointer access
```

Shift Operations

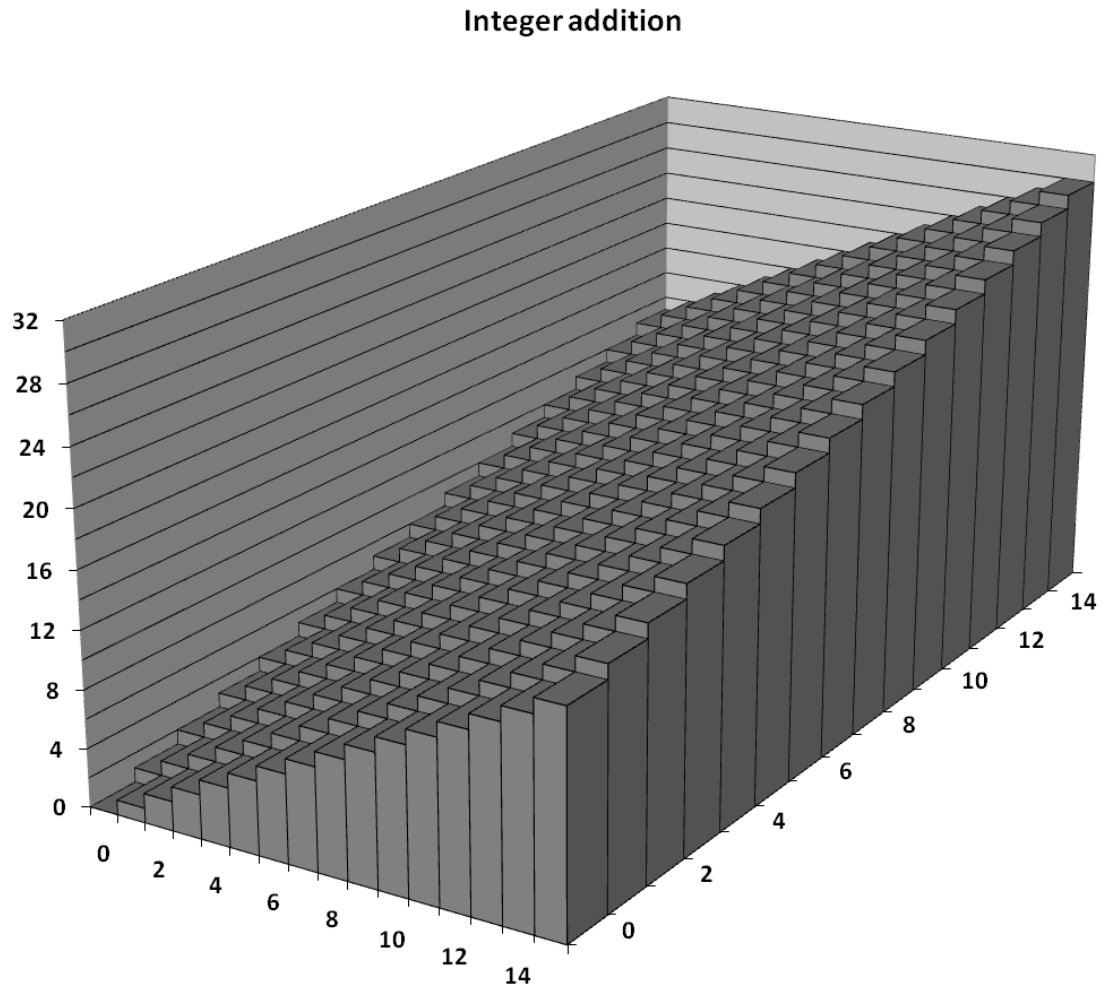
- Left shift: ***x << y***
  - Shift bit-vector ***x*** left ***y*** positions
    - Throw away extra bits on left
    - Fill with ***0***'s on right
- Right shift: ***x >> y***
  - Shift bit-vector ***x*** right ***y*** positions
    - Throw away extra bits on right
  - Logical shift: fill with ***0***'s on left
  - Arithmetic shift: replicate MSB on right
    - Useful with two's complement integer representation
- Undefined if ***y < 0*** or ***y*** ≥ 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

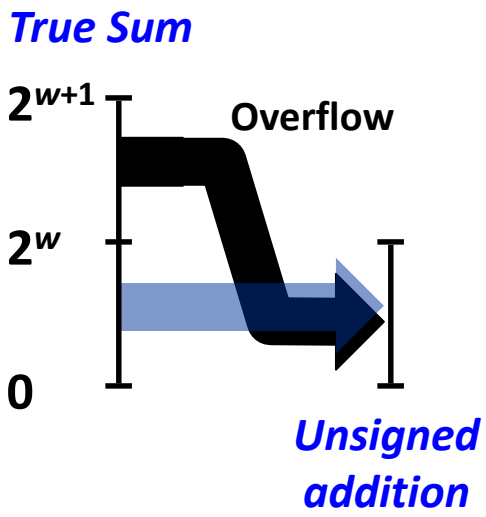
Addition (1)

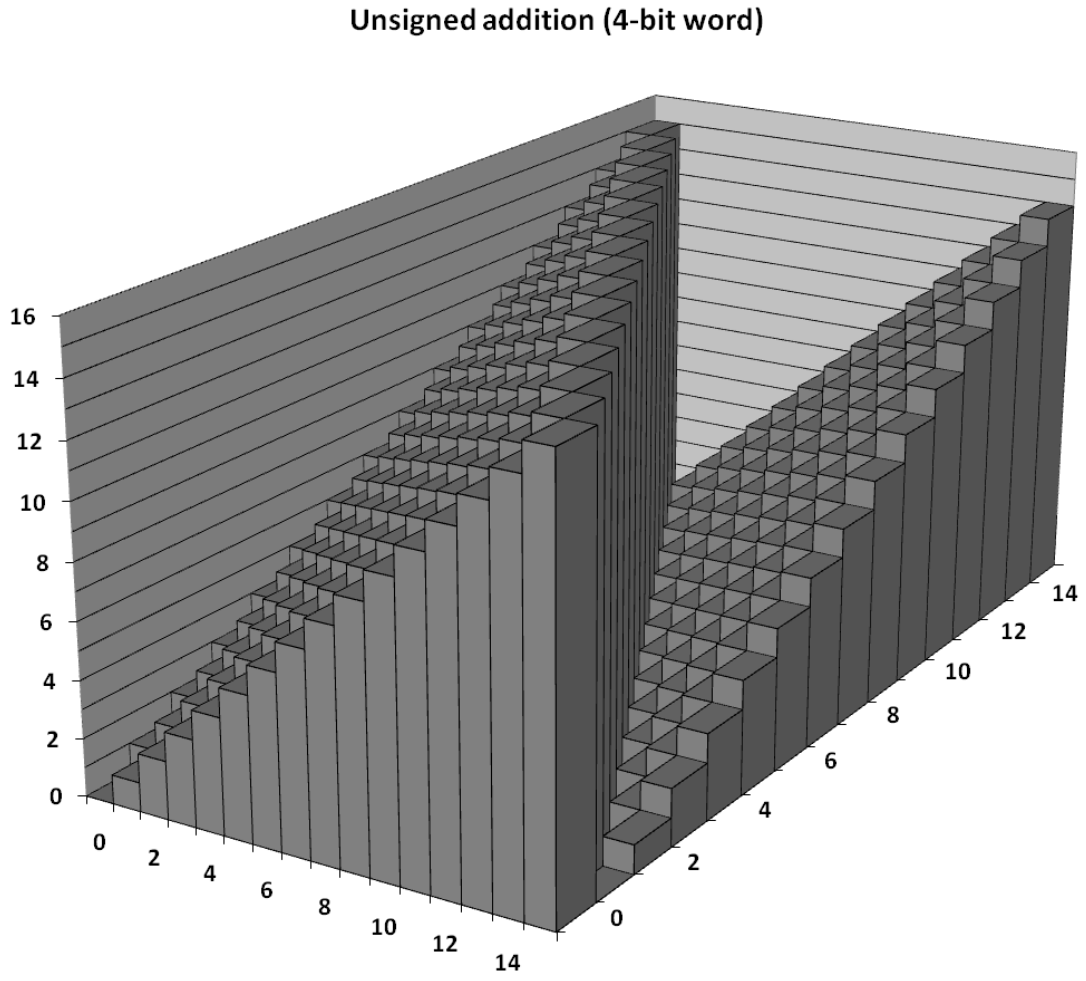
- Integer addition example
  - 4-bit integers ***u***, ***v***
  - Compute true sum
  - True sum requires one more bit ("carry")
  - Values increase linearly with ***u*** and ***v***
  - Forms planar surface



Addition (2)

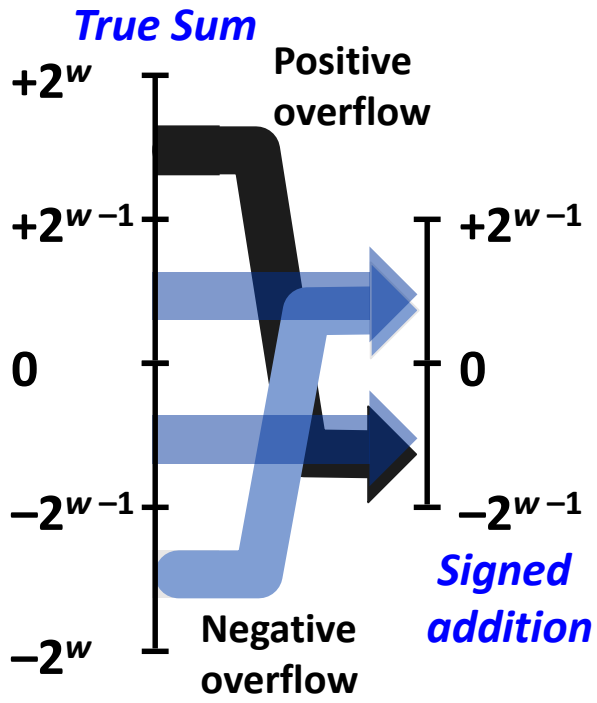
- Unsigned addition
  - Ignores carry output
  - Wraps around
    - If true sum ≥ 2^w
    - At most once

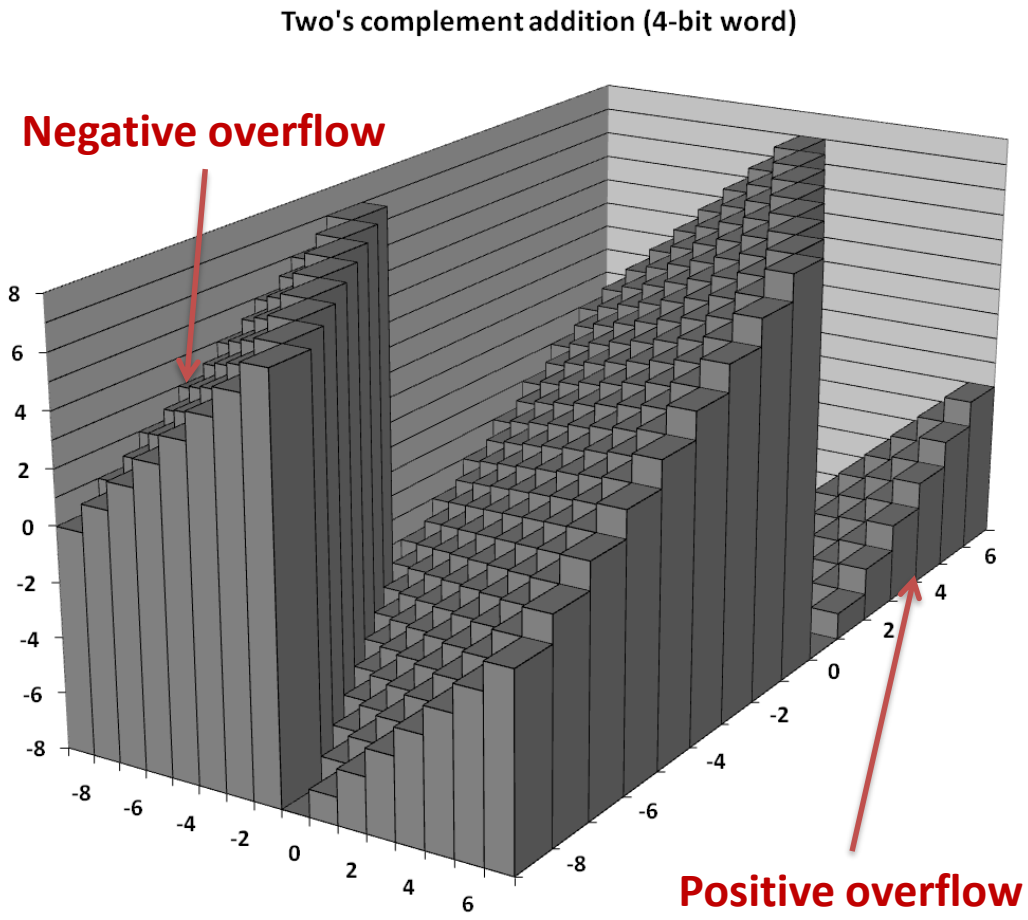




Addition (3)

- Signed addition
  - Drop off MSB
  - Treat remaining bits as two's complement integers





Addition (4)

- Signed addition in C
  - Ignores carry output
  - The low-order ***w*** bits are identical to unsigned addition

Examples for w = 3

Mode	x	y	x + y	Truncated x + y
Unsigned	4 [100]	3 [011]	7 [0111]	7 [111]
Two's comp.	-4 [100]	3 [011]	-1 [1111]	-1 [111]
Unsigned	4 [100]	7 [111]	11 [1011]	3 [011]
Two's comp.	-4 [100]	-1 [111]	-5 [1011]	3 [011]
Unsigned	3 [011]	3 [011]	6 [0110]	6 [110]
Two's comp.	3 [011]	3 [011]	6 [0110]	-2 [110]

Multiplication (1)

- Ranges of (x * y)
  - Unsigned: up to ***2w*** bits

$$0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$$

  - Two's complement min: up to ***2w-1*** bits

$$x * y \ge (-2^{w-1}) * (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$$

  - Two's complement max: up to ***2w*** bits (only for $TMin^2$)

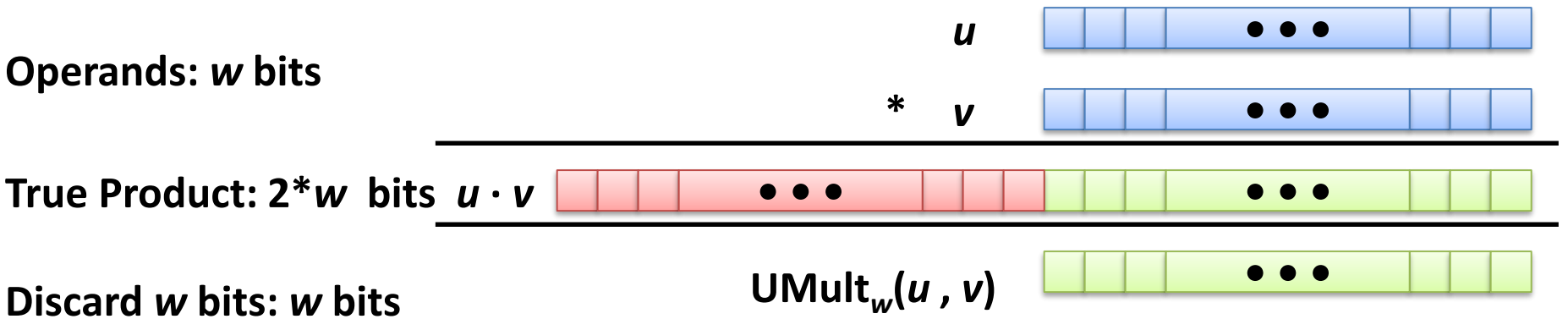
$$x * y \le (-2^{w-1})^2 = 2^{2w-2}$$

- Maintaining exact results
  - Would need to keep expanding word size with each product computed
  - Done in software by "arbitrary precision" arithmetic packages

Multiplication (2)

- Unsigned multiplication in C
  - Ignores high order ***w*** bits
  - Implements modular arithmetic

$$UMult_w(u, v) = u \cdot v \mod 2^w$$



Multiplication (3)

- Signed multiplication in C
  - Ignores high order ***w*** bits
  - The low-order ***w*** bits are identical to unsigned multiplication

Examples for w = 3

Mode	x	y	x · y	Truncated x · y
Unsigned	5 [101]	3 [011]	15 [001111]	7 [111]
Two's comp.	-3 [101]	3 [011]	-9 [110111]	-1 [111]
Unsigned	4 [100]	7 [111]	28 [011100]	4 [100]
Two's comp.	-4 [100]	-1 [111]	4 [000100]	-4 [100]
Unsigned	3 [011]	3 [011]	9 [001001]	1 [001]
Two's comp.	3 [011]	3 [011]	9 [001001]	1 [001]

Multiplication (4)

- Power-of-2 multiply with shift
  - $u << k$ gives $u * 2^k$
    - e.g. $u << 3 == u * 8$, $(u << 5) - (u << 3) == u * 24$
  - Both signed and unsigned
  - Most machines shift and add faster than multiply