

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

Systems Software &  
Architecture Lab.

Seoul National University

Fall 2018

# Integers



# Computer Systems



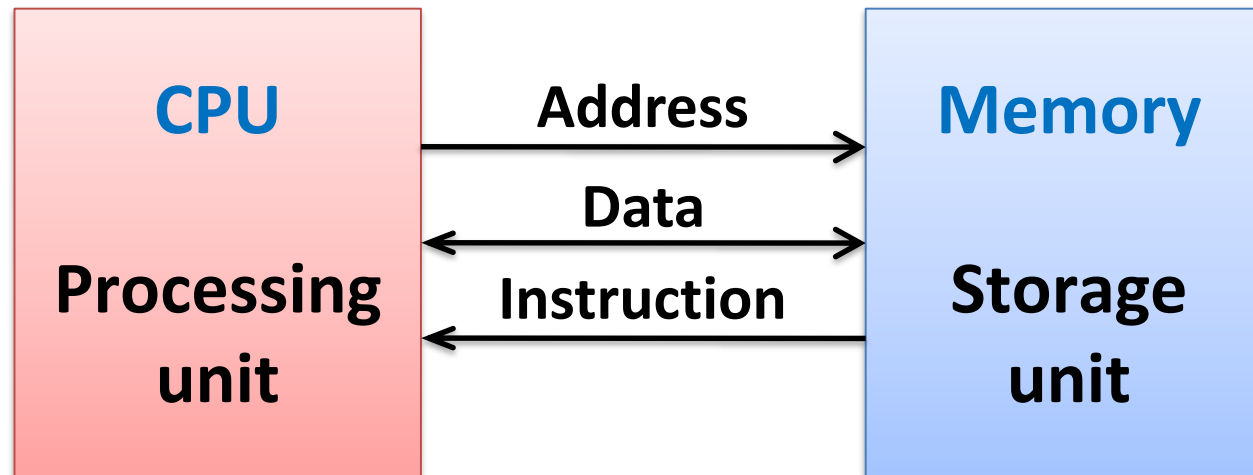
Google TV™



A computer is a  machine.

# Von Neumann Architecture

- By John von Neumann, 1945

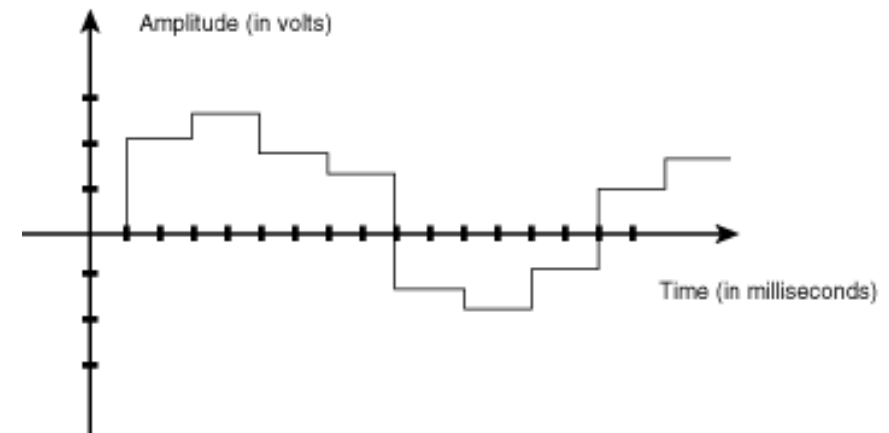
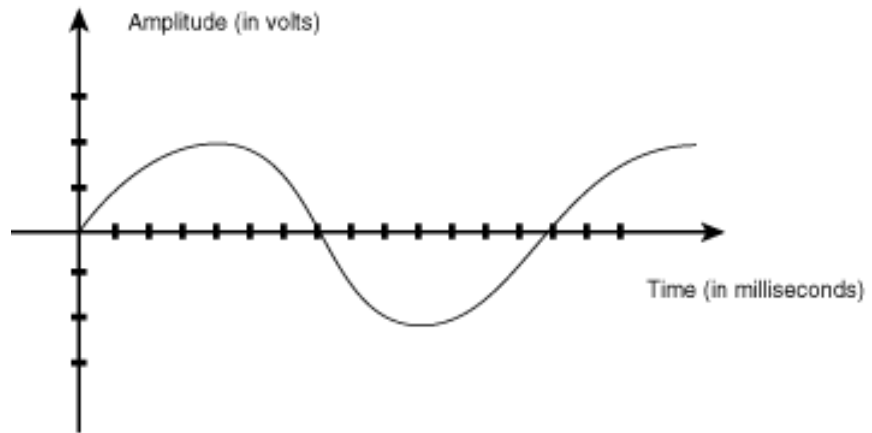


Data movement  
Arithmetic & logical ops  
Control transfer

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

# The Advent of the Digital Age

- Analog vs. digital?



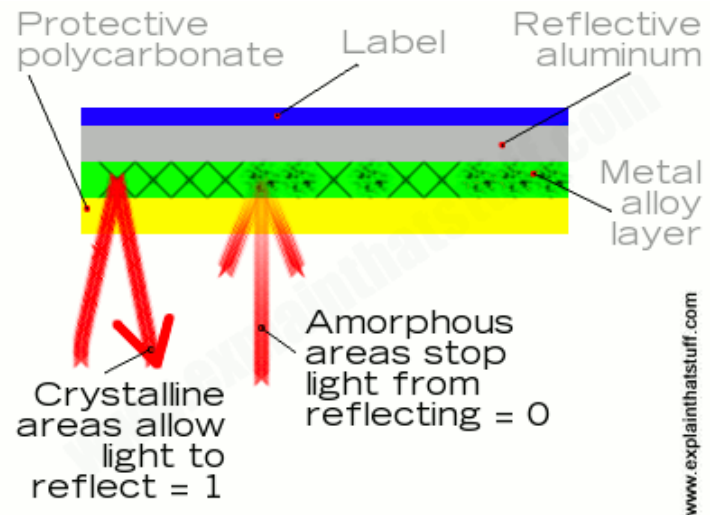
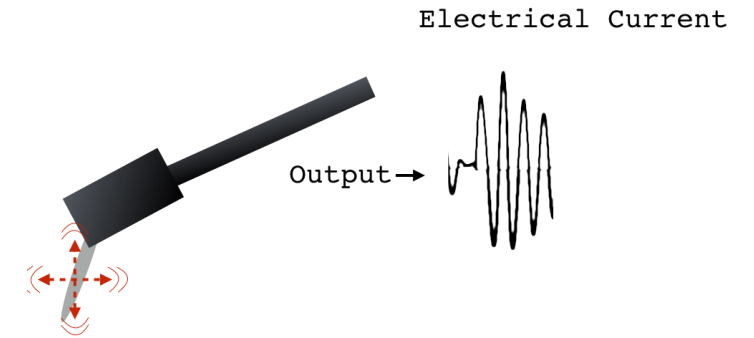
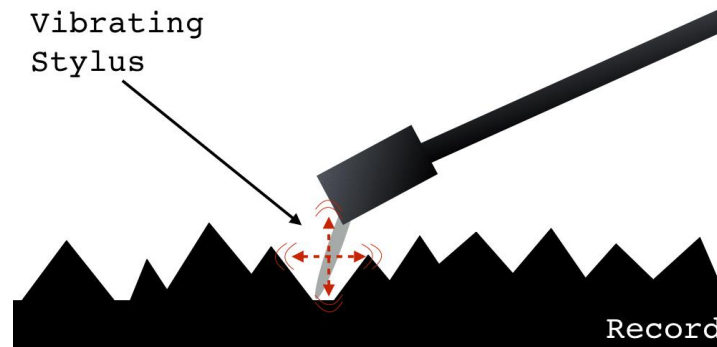
- Compact Disc (CD)

- 44.1 KHz, 16-bit, 2-channel

- MP3

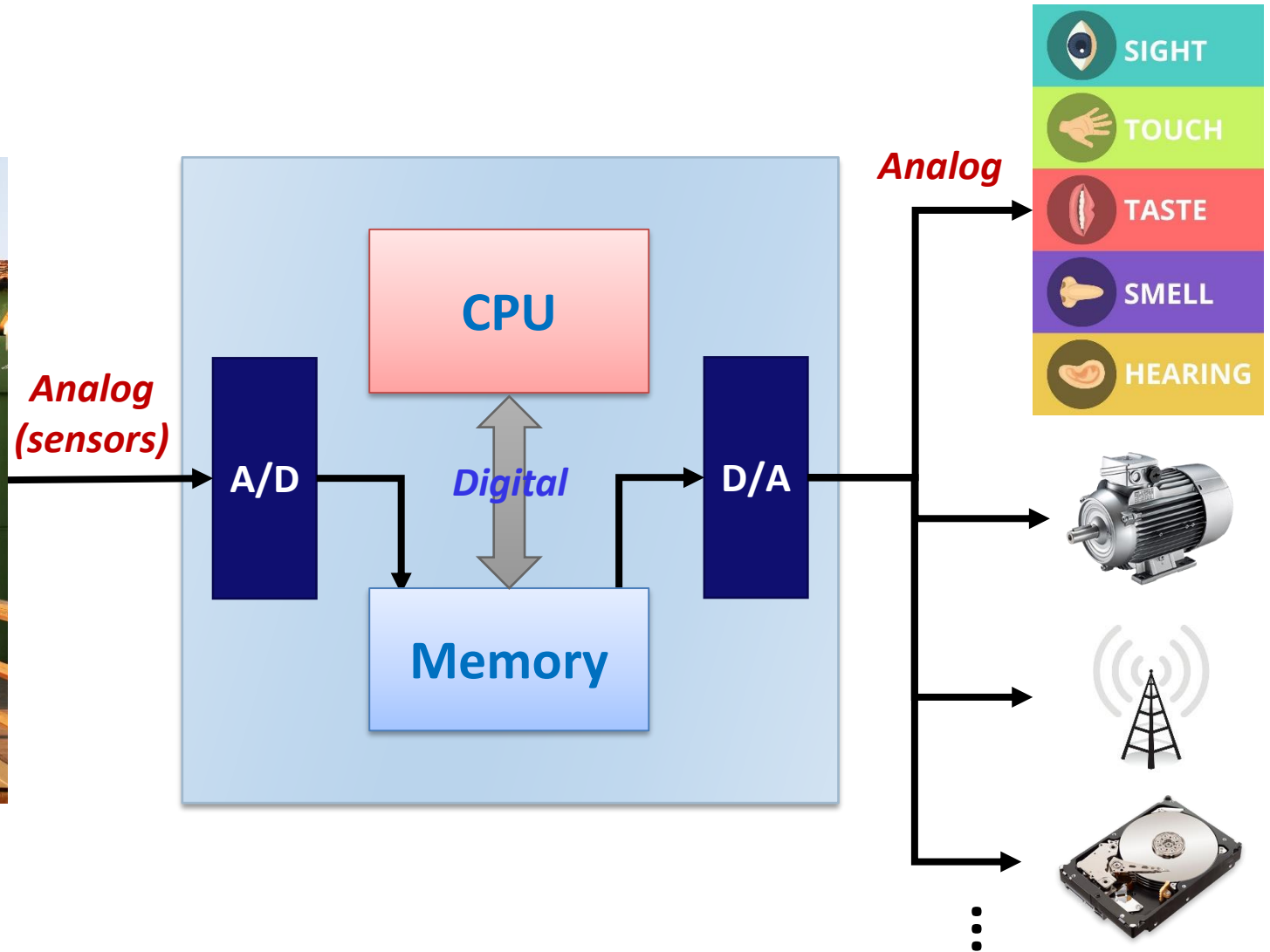
- A digital audio encoding with lossy data compression

# 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

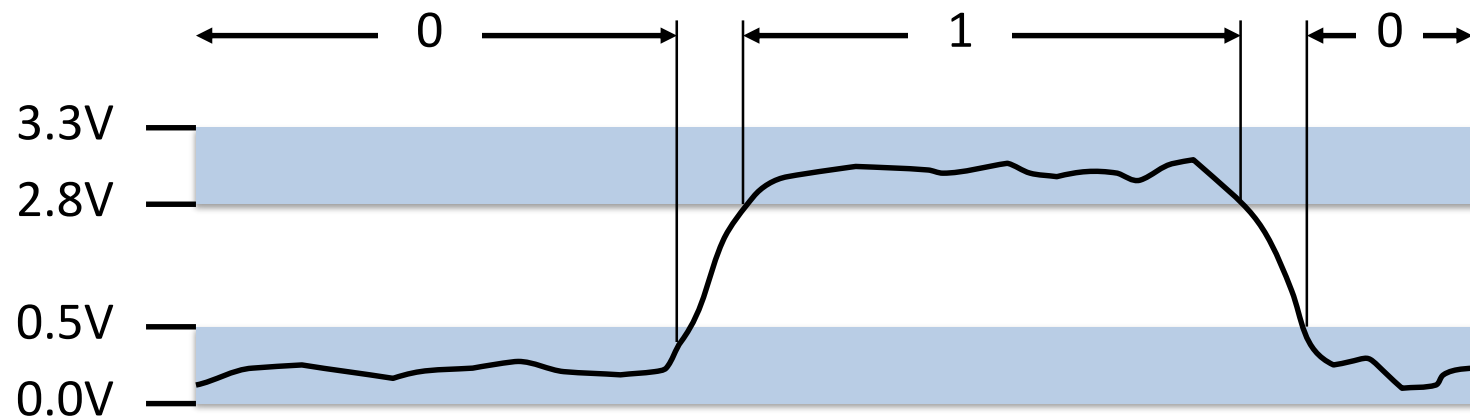
- Computers manipulate representations of things
- Things are represented as binary digits
- What can you represent with  $N$  bits?
  - $2^N$  things
  - Numbers, characters, pixels, positions, source code, executable files, machine instructions, ...
  - Depends on what operations you do on them

	01010011 01001110 01010101 01000001 01000010 01000011 01001000 01001001							
(char)	'S'	'N'	'U'	'A'	'R'	'C'	'H'	'I'
(int)	1096109651				1229472594			
(double)	1.08216479583800794... x 10 <sup>45</sup>							



# 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

$$B = [b_{w-1}, b_{w-2}, \dots, b_0] \quad \mathbf{x = 0001\ 0000\ 0101\ 1110}_2$$

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

$$\begin{aligned} \mathbf{D(x)} &= \mathbf{2^{12} + 2^6 + 2^4 + 2^3 + 2^2 + 2^1} \\ &= \mathbf{4096 + 64 + 16 + 8 + 4 + 2} \\ &= \mathbf{4190} \end{aligned}$$

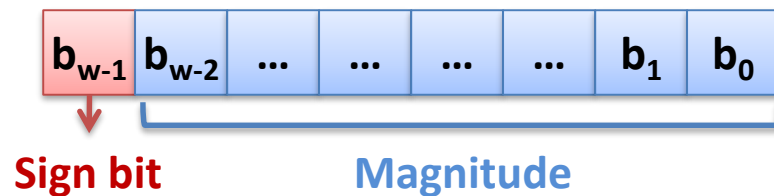
- What is the range for unsigned values with  $w$  bits?

# Signed Integers

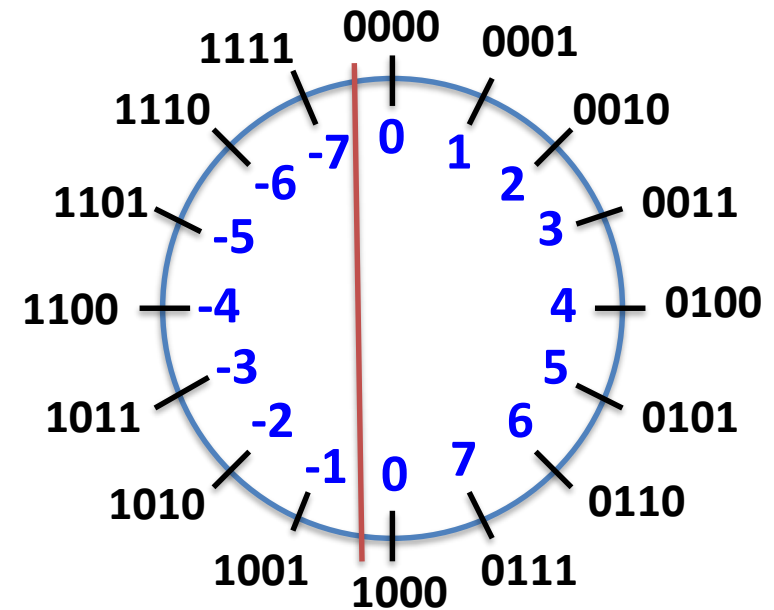
- Encoding positive numbers
  - Same as unsigned numbers
- Encoding negative numbers
  - Sign-magnitude representation
  - Ones' complement representation
  - Two's complement representation

# Sign-magnitude Representation

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



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

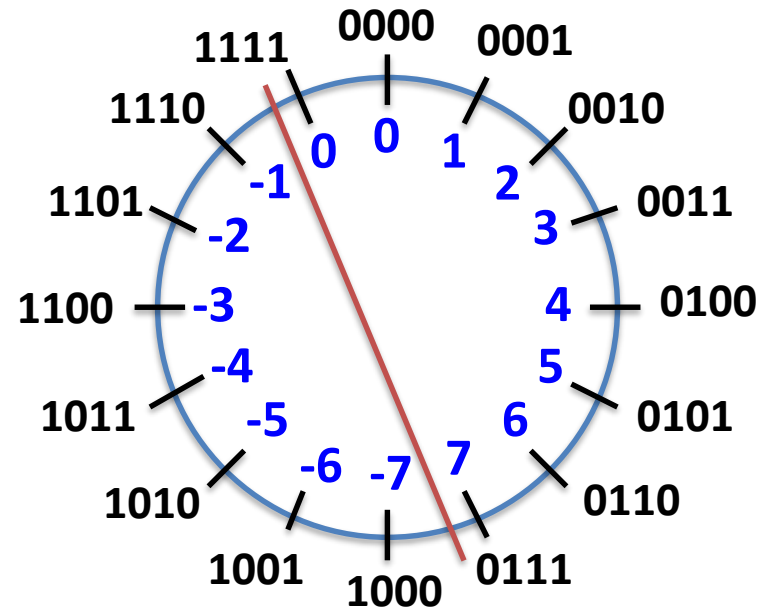


# Ones' Complement Representation

- Easy to find  $-n$
- Two zeros
  - $[000\dots00]$ ,  $[111\dots11]$
- No longer used



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

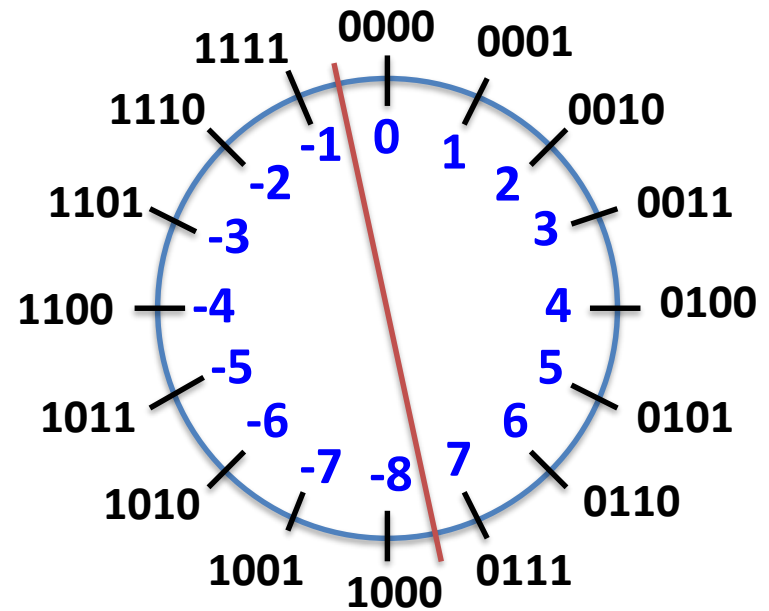


# Two's Complement Representation (I)

- Unique zero
- Easy for hardware
  - leading 0  $\geq 0$
  - 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:  $\sim x + x == 1111\dots11_2 == -1$

- Increment

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

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

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

# Numeric Ranges (I)

- Unsigned values

- $U_{Min} = 0$  [000...00]
- $U_{Max} = 2^w - 1$  [111...11]

- Two's complement values

- $T_{Min} = -2^{w-1}$  [100...00]
- $T_{Max} = 2^{w-1} - 1$  [011...11]

Values for  $w = 16$

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

# Numeric Ranges (2)

- Values for different word sizes

	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

- Observations

- $|TMin| = TMax + 1$   
(Asymmetric range)
- $UMax = 2 * TMax + 1$

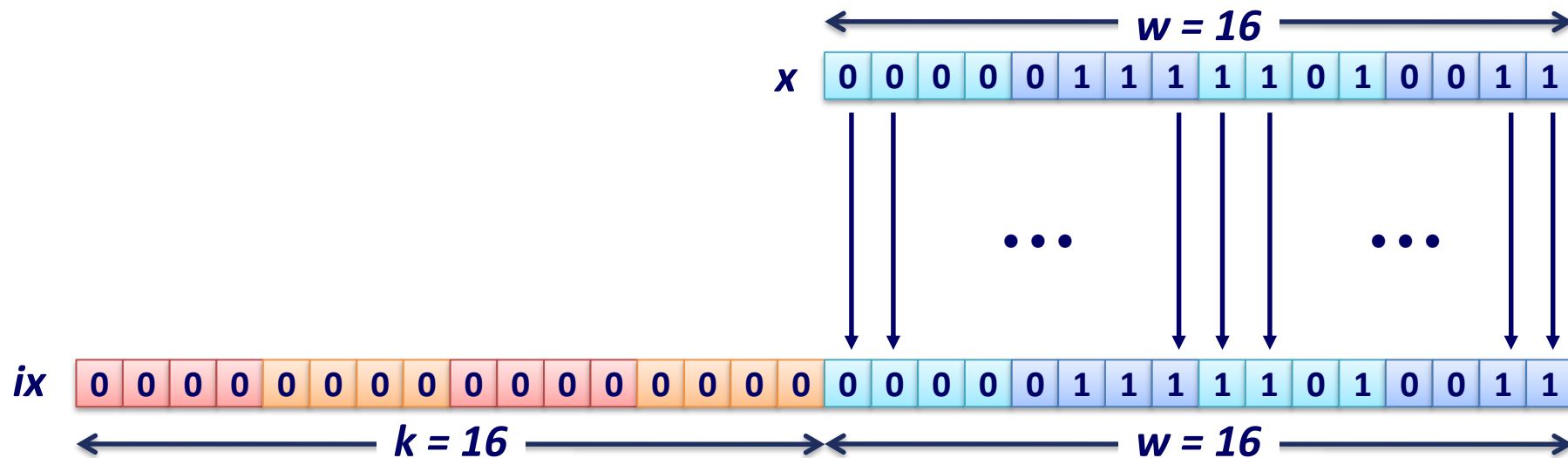
- In C programming

- `#include <limits.h>`
- `INT_MIN`, `INT_MAX`,  
`LONG_MIN`, `LONG_MAX`,  
`UINT_MAX`, ...
- Values platform-specific

# Type Conversion (I)

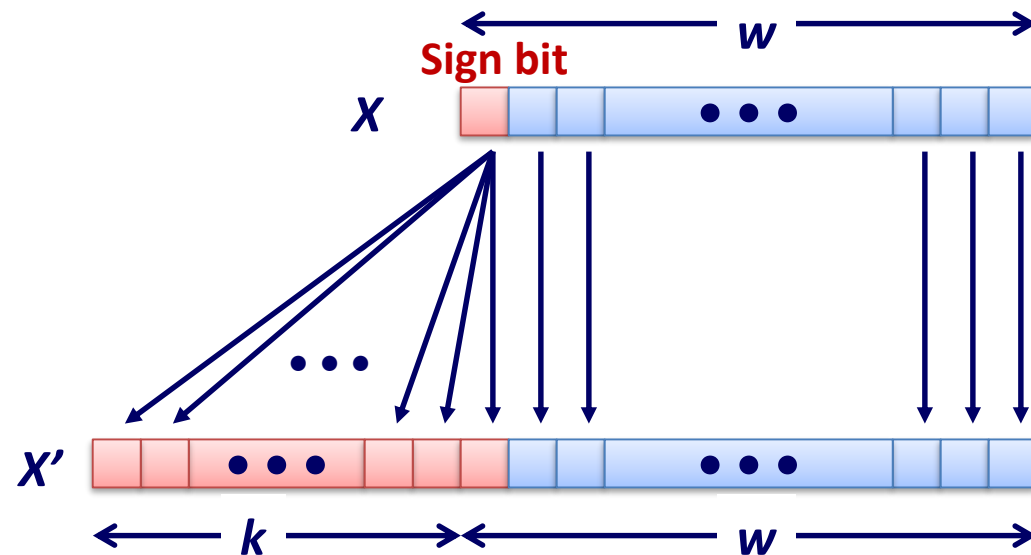
- Unsigned:  $w$  bits  $\rightarrow w+k$  bits
  - Zero extension: just fill  $k$  bits with 0's

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



# Type Conversion (2)

- Signed:  $w$  bits  $\rightarrow$   $w+k$  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)

- Sign extension example
  - Converting from smaller to larger integer type
  - C automatically performs sign extension

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

	Decimal	Hex	Binary
x	2003	07 D3	00000111 11010011
ix	2003	00 00 07 D3	<b>00000000 00000000</b> 00000111 11010011
y	-2003	F8 2D	11111000 00101101
iy	-2003	FF FF F8 2D	<b>11111111 11111111</b> 11111000 00101101

# Type Conversion (4)

- Unsigned & Signed:  $w+k$  bits  $\rightarrow$   $w$  bits
  - Just truncate it to lower  $w$  bits
  - Equivalent to computing  $x \bmod 2^w$

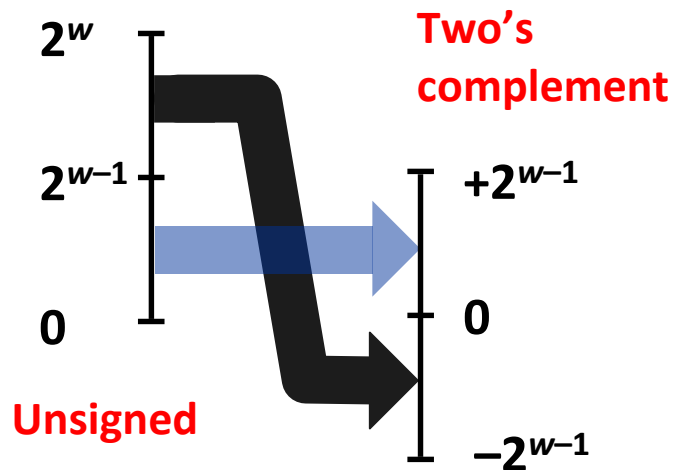
```
unsigned int    x = 0xcafebabe;
unsigned short ix = (unsigned short) x;
int            y = 0x2003beef;
short         iy = (short) y;
```

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

# Type Conversion (5)

- 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;
    
```

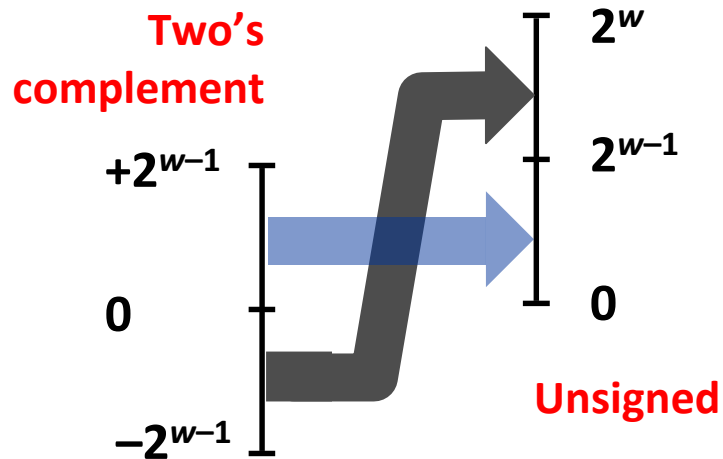
	Decimal	Hex	Binary
<b>x</b>	2003	07 D3	00000111 11010011
<b>ix</b>	2003	07 D3	00000111 11010011
<b>y</b>	47806	BA BE	10111010 10111110
<b>iy</b>	-17730	BA BE	10111010 10111110



# Type Conversion (6)

- 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
<code>0 == 0U</code>	unsigned	True
<code>-1 &lt; 0</code>	signed	True
<code>-1 &lt; 0U</code>	unsigned	?
<code>-1 &gt; -2</code>	signed	?
<code>(unsigned) -1 &gt; -2</code>	unsigned	?
<code>2147483647 &gt; -2147483647-1</code>	signed	?
<code>2147483647U &gt; -2147483647-1</code>	unsigned	?
<code>2147483647 &gt; (int) 2147483648U</code>	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;
}
```

# Example 4

```
int sum_array (int a[], unsigned len)
{
    int i;
    int result = 0;

    for (i = 0; i <= len - 1; i++)
        result += a[i];

    return result;
}
```

# 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)

<code>~0x41 → 0xBE</code>	<code>~01000001<sub>2</sub> → 10111110<sub>2</sub></code>
<code>~0x00 → 0xFF</code>	<code>~00000000<sub>2</sub> → 11111111<sub>2</sub></code>
<code>0x69 &amp; 0x55 → 0x41</code>	<code>01101001<sub>2</sub> &amp; 01010101<sub>2</sub> → 01000001<sub>2</sub></code>
<code>0x69   0x55 → 0x7D</code>	<code>01101001<sub>2</sub> &amp; 01010101<sub>2</sub> → 01111101<sub>2</sub></code>
<code>0x69 ^ 0x55 → 0x3C</code>	<code>01101001<sub>2</sub> &amp; 01010101<sub>2</sub> → 00111100<sub>2</sub></code>

# Logic Operations in C

- `&&`, `||`, `!`
  - View 0 as “False”, anything nonzero as “True”
  - Always return 0 or 1
  - Early termination
- Examples (char data type)

<code>!0x41</code>	<code>→</code>	<code>0x00</code>
<code>!0x00</code>	<code>→</code>	<code>0x01</code>
<code>!!0x41</code>	<code>→</code>	<code>0x01</code>
<code>0x69 &amp;&amp; 0x55</code>	<code>→</code>	<code>0x01</code>
<code>0x69    0x55</code>	<code>→</code>	<code>0x01</code>
<code>if (p &amp;&amp; *p) ...</code>		<code>// avoids null pointer access</code>

# Shift Operations

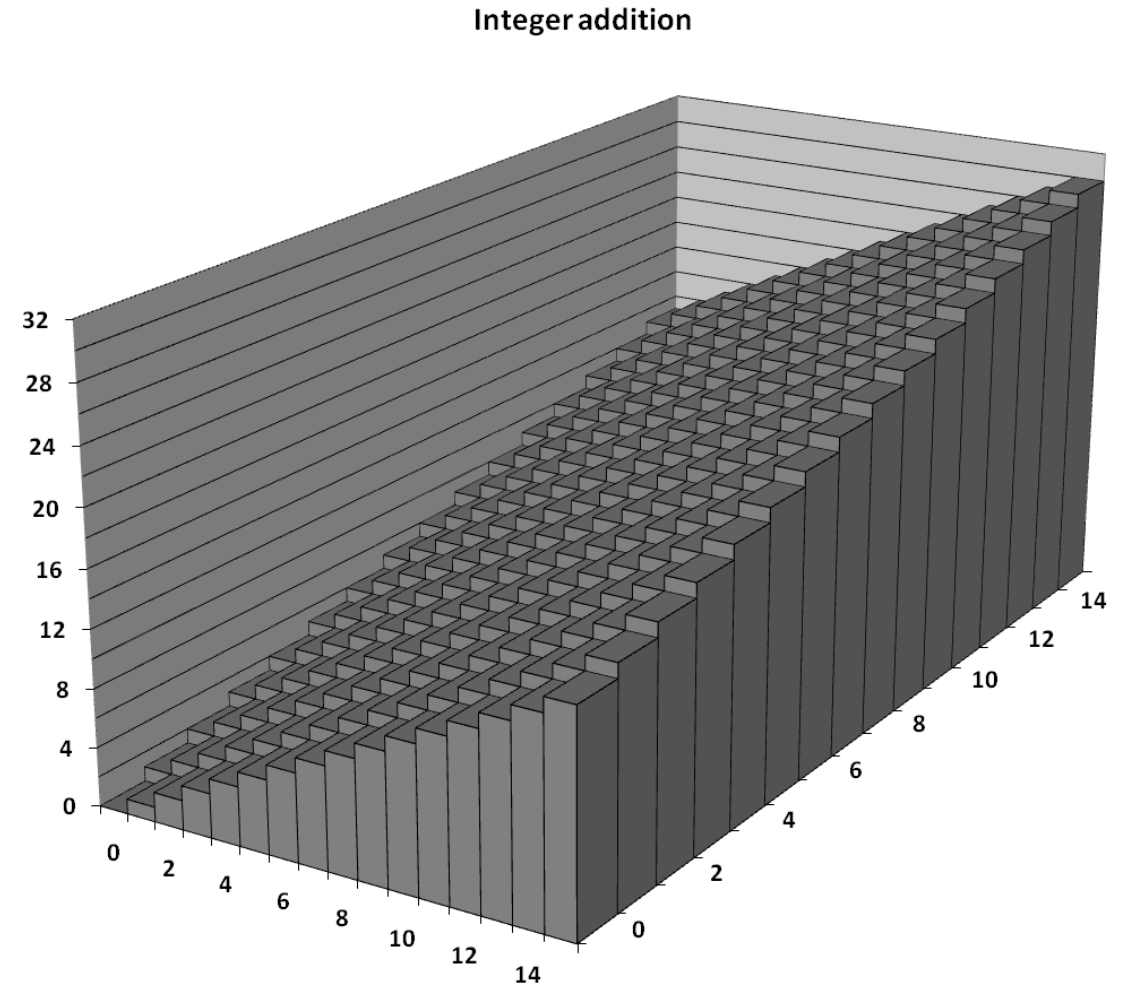
- **Left shift:**  $x \ll y$ 
  - Shift bit-vector  $x$  left  $y$  positions
    - Throw away extra bits on left
    - Fill with  $0$ 's on right
- **Right shift:**  $x \gg y$ 
  - Shift bit-vector  $x$  right  $y$  positions
    - Throw away extra bits on right
  - Logical shift: fill with  $0$ 's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

Argument x	01100010
$\ll 3$	00010000
Log. $\gg 2$	00011000
Arith. $\gg 2$	00011000

Argument x	10100010
$\ll 3$	00010000
Log. $\gg 2$	00101000
Arith. $\gg 2$	11101000

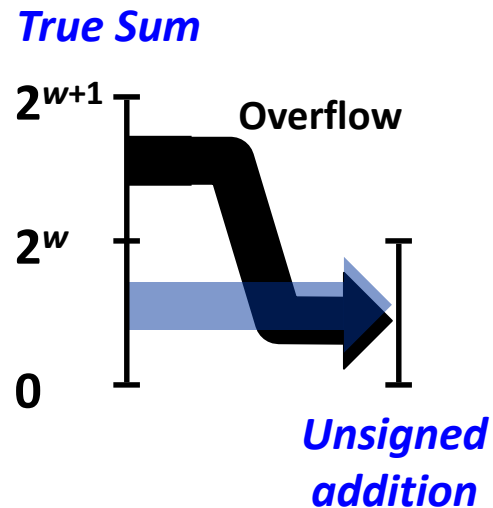
# Addition (I)

- 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

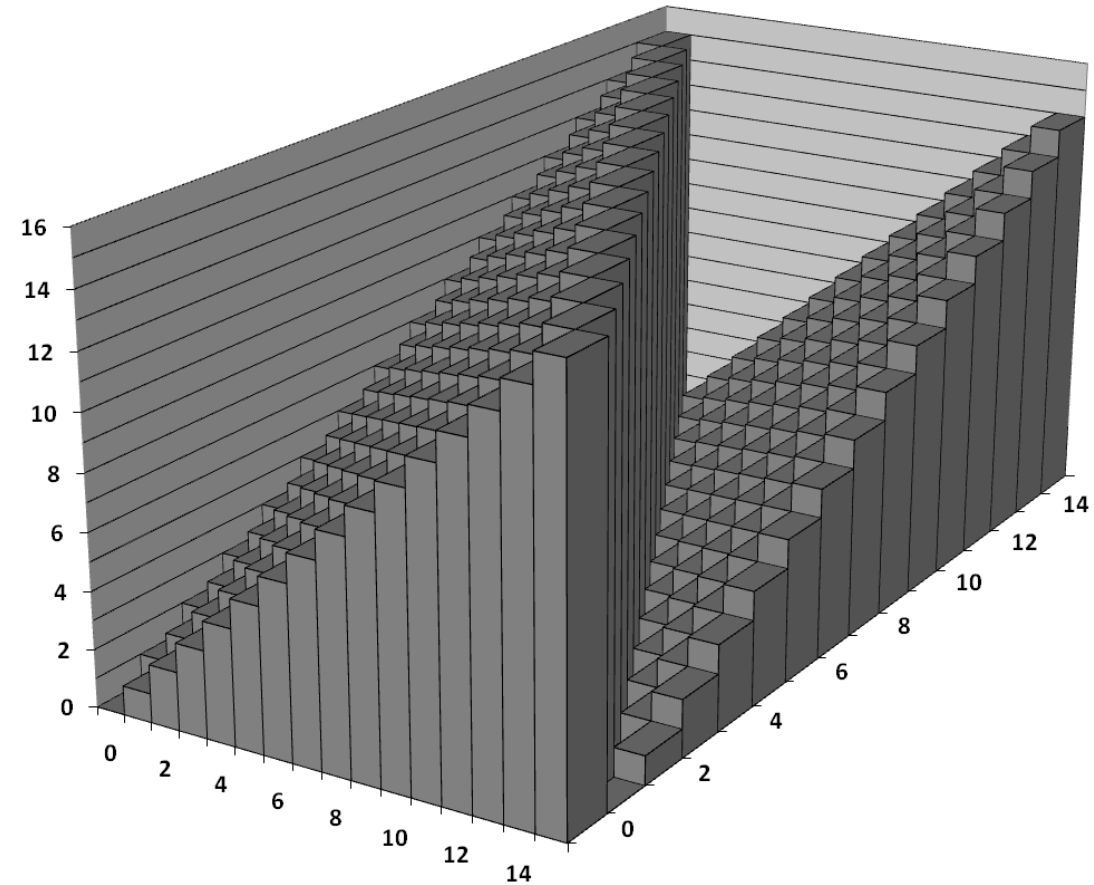


# Addition (2)

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



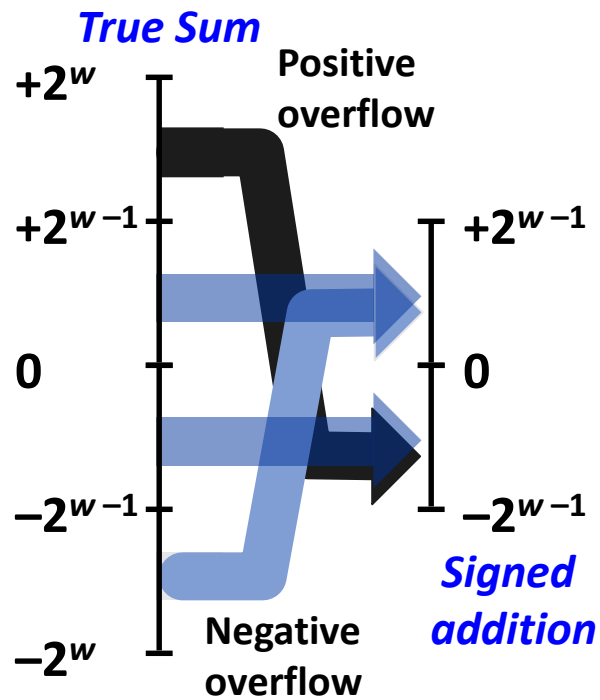
Unsigned addition (4-bit word)



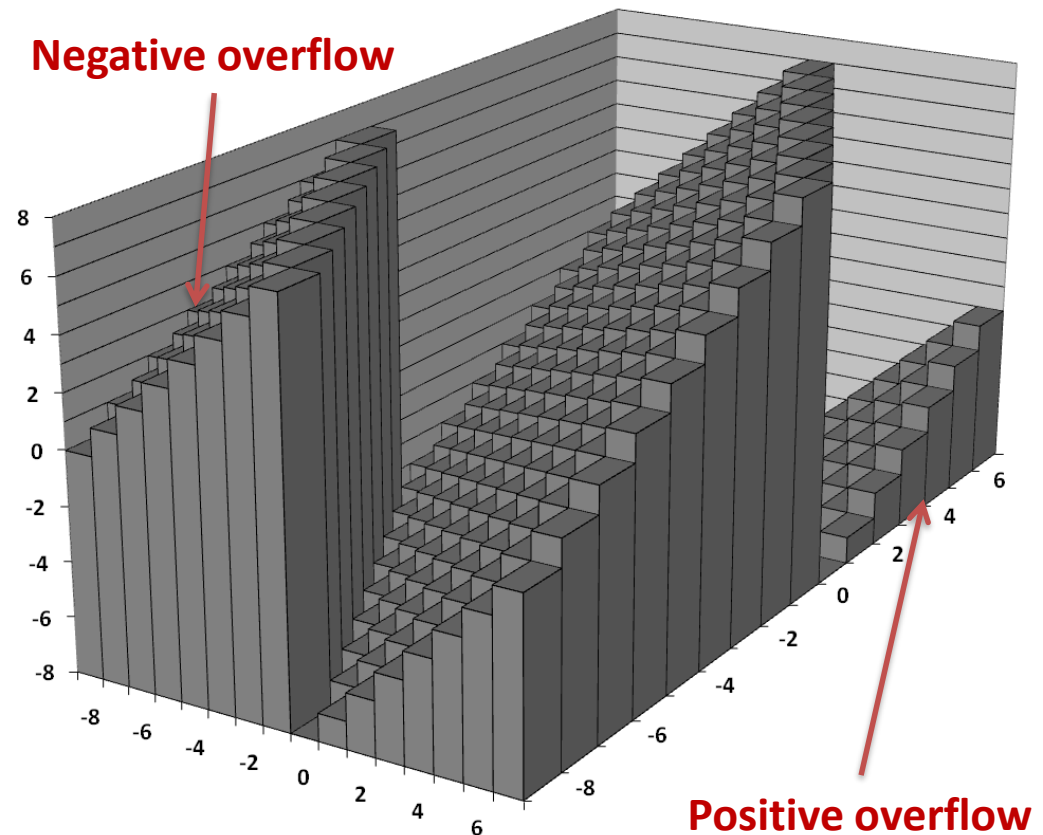
# Addition (3)

- Signed addition

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



Two's complement addition (4-bit word)



# 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 (I)

## ■ Ranges of $(x * y)$

- Unsigned: up to  $2w$  bits

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

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

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

- Two's complement max: up to  $2w$  bits (only for TMin<sup>2</sup>)

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

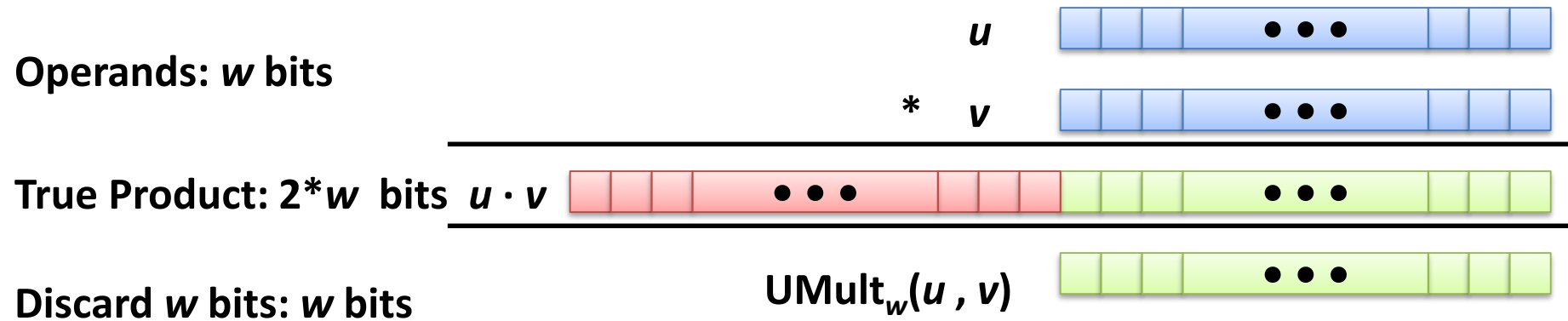
## ■ Maintaining exact results

- 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

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



# Multiplication (3)

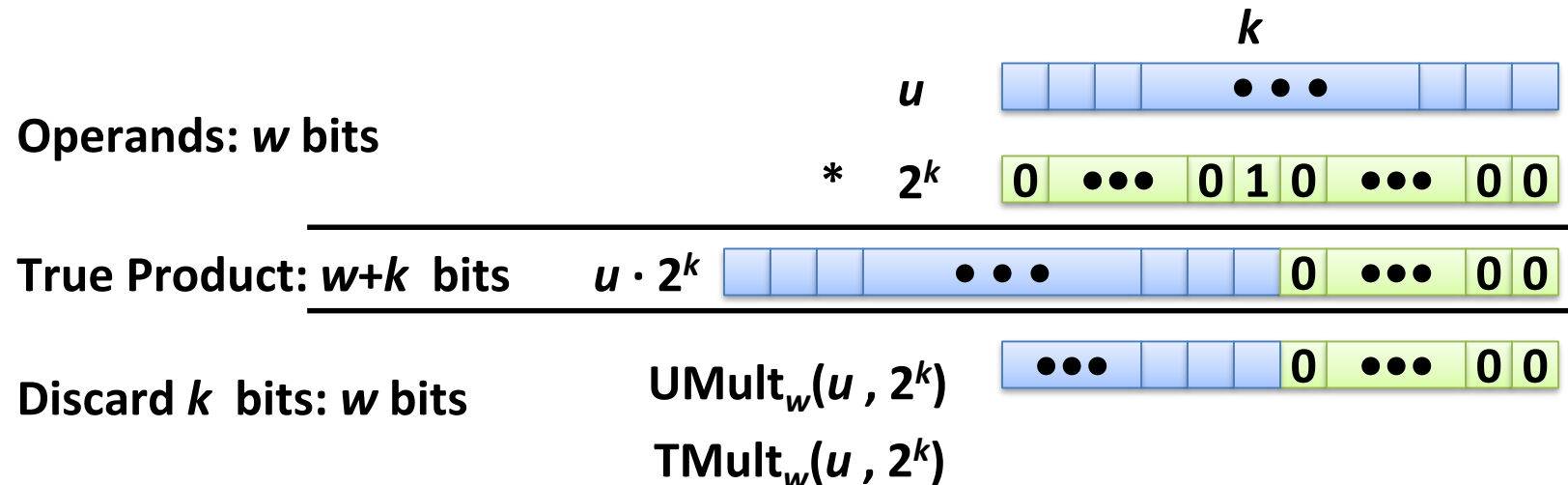
- Signed multiplication in C
  - Ignores high order  $w$  bits
  - The low-order  $w$  bits are identical to unsigned multiplication

## Examples for $w = 3$

Mode	x	y	$x \cdot y$	Truncated $x \cdot y$
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
  - $u \ll k$  gives  $u * 2^k$ 
    - e.g.  $u \ll 3 == u * 8$ ,  $(u \ll 5) - (u \ll 3) == u * 24$
  - Both signed and unsigned
  - Most machines shift and add faster than multiply



# Multiplication (5)

- Compiled multiplication code

- C compiler automatically generates shift/add code when multiplying by constant

## C Function

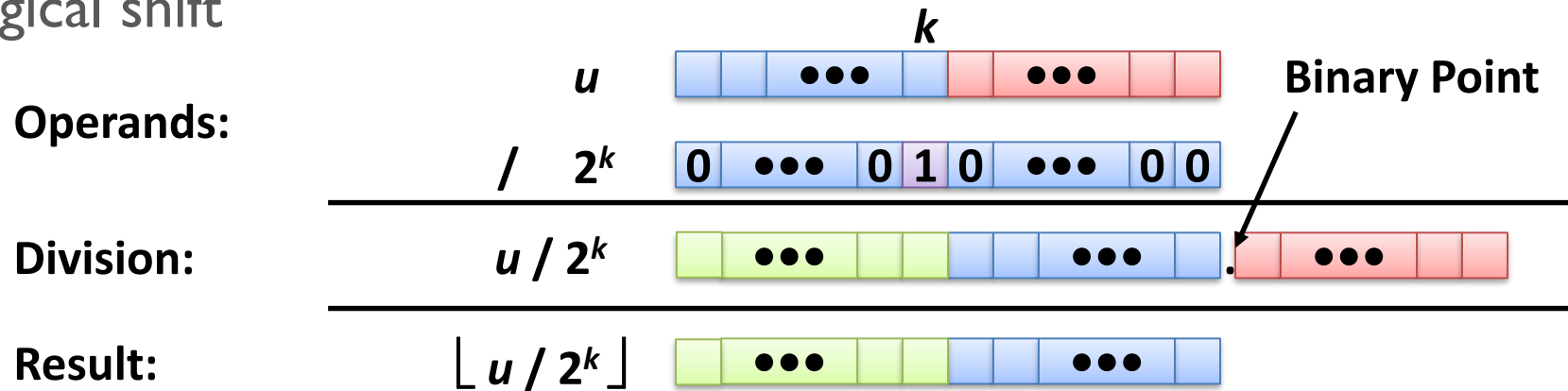
```
int mul12 (int x)
{
    return x * 12;
}
```

## Compiled Arithmetic Operations

```
leal    (%eax, %eax, 2), %eax    ; t ← x + x * 2
sall    $2, %eax                ; return t << 2
```

# Division (I)

- Unsigned power-of-2 divide with shift
  - $u \gg k$  gives  $\lfloor u / 2^k \rfloor$
  - Uses logical shift



Expression	Division	Result	Hex	Binary
$x$	15213	15213	3B 6D	00111011 01101101
$x \gg 1$	7606.5	7606	1D B6	00011101 10110110
$x \gg 4$	950.8125	950	03 B6	00000011 10110110
$x \gg 8$	59.4257813	59	00 3B	00000000 00111011

# Division (2)

- Compiled unsigned division code
  - Uses logical shift for unsigned
  - Logical shift written as >>> in Java

## C Function

```
unsigned udiv8 (unsigned x)
{
    return x / 8;
}
```

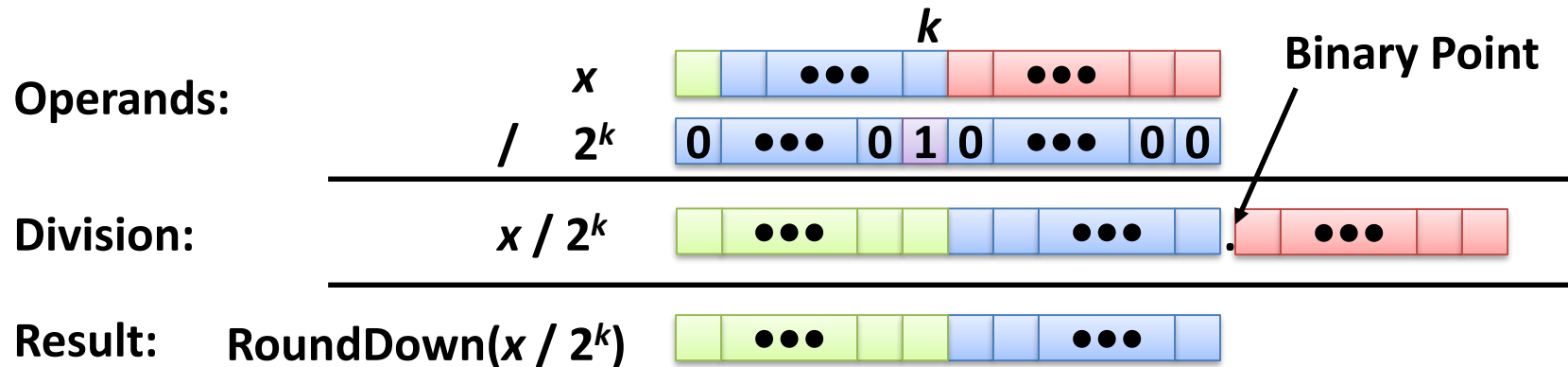
## Compiled Arithmetic Operations

```
shrl    $3, %eax                ; return t >> 3
```

# Division (3)

- Signed power-of-2 divide with shift

- $x \gg k$  gives  $\lfloor x / 2^k \rfloor$
- Uses arithmetic shift (rounds wrong direction if  $x < 0$ )

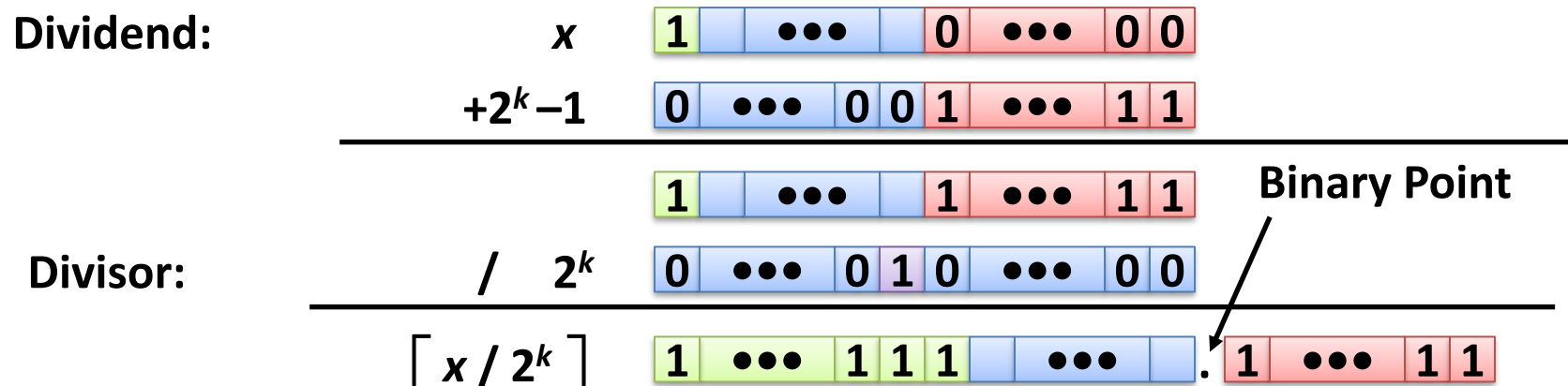


Expression	Division	Result	Hex	Binary
$y$	-15213	-15213	C4 93	11000100 10010011
$y \gg 1$	-7606.5	-7607	E2 49	<b>1</b> 1100010 01001001
$y \gg 4$	-950.8125	-951	FC 49	<b>1111</b> 1100 01001001
$y \gg 8$	-59.4257813	-60	FF C4	<b>11111111</b> 11000100



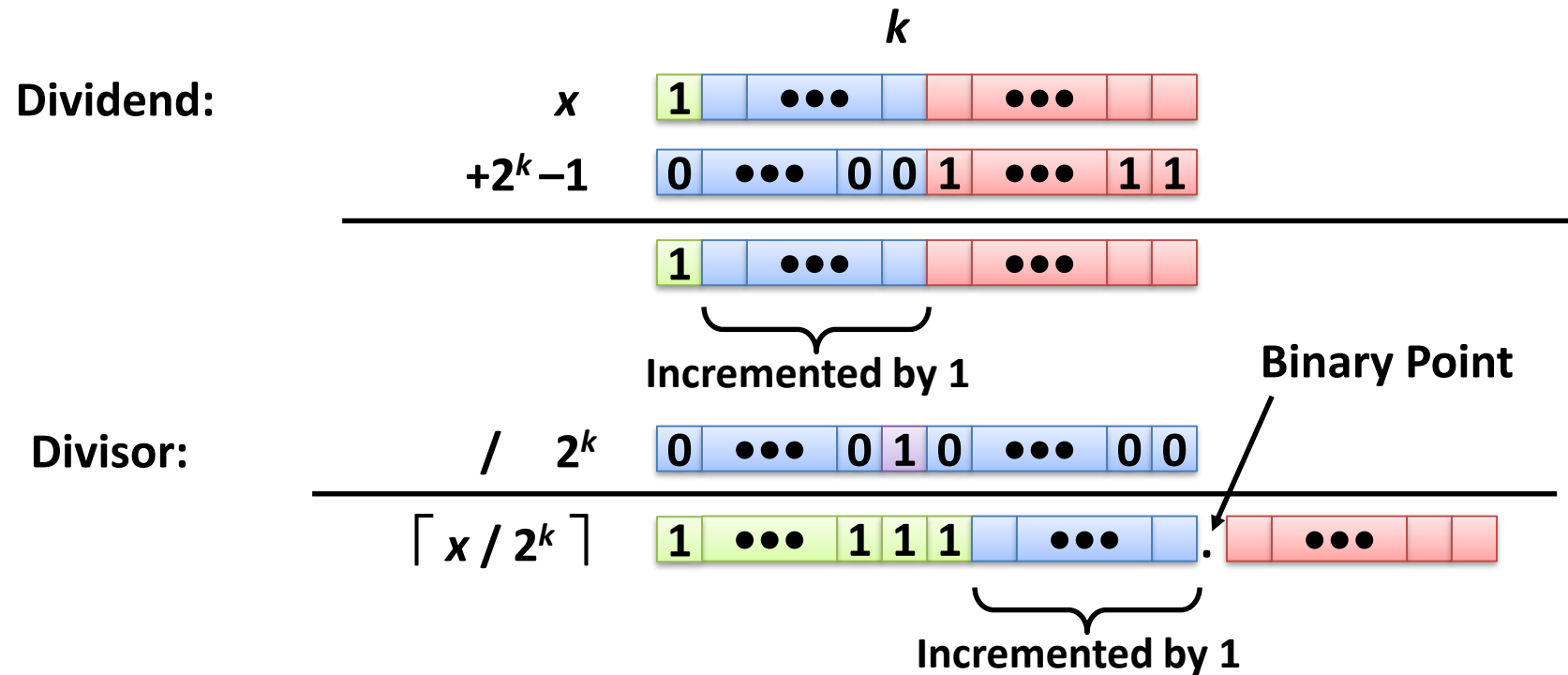
# Division (4)

- Correct power-of-2 divide
  - Want  $\lceil x / 2^k \rceil$  (Round Toward 0) when  $x < 0$
  - Compute as  $\lfloor (x + 2^k - 1) / 2^k \rfloor$ 
    - In C:  $(x + (1 \ll k) - 1) \gg k$
    - Biases dividend toward 0
- Case 1: No rounding
  - Biasing has no effect



# Division (5)

- Case 2: Rounding
  - Biasing adds 1 to final result



# Division (6)

- Compiled signed division code
  - Uses arithmetic shift for signed
  - Arithmetic shift written as `>>` in Java

## Compiled Arithmetic Operations

```
L3:    testl    %eax, %eax
      js     L4
      sarl   $3, %eax
      ret
L4:    addl   $7, %eax
      jmp   L3
```

## C Function

```
int idiv8 (int x)
{
    return x / 8;
}
```

## Explanation

```
if (x < 0)
    x += 7;
return x >> 3;
```