Integer Representation

Prof. Hyuk-Yoon Kwon
http://bigdata.seoultech.ac.kr

Most parts are based on slides written by Brayant and O'Hallaon, CMU
(http://csapp.cs.cmu.edu/3e/instructors.html)

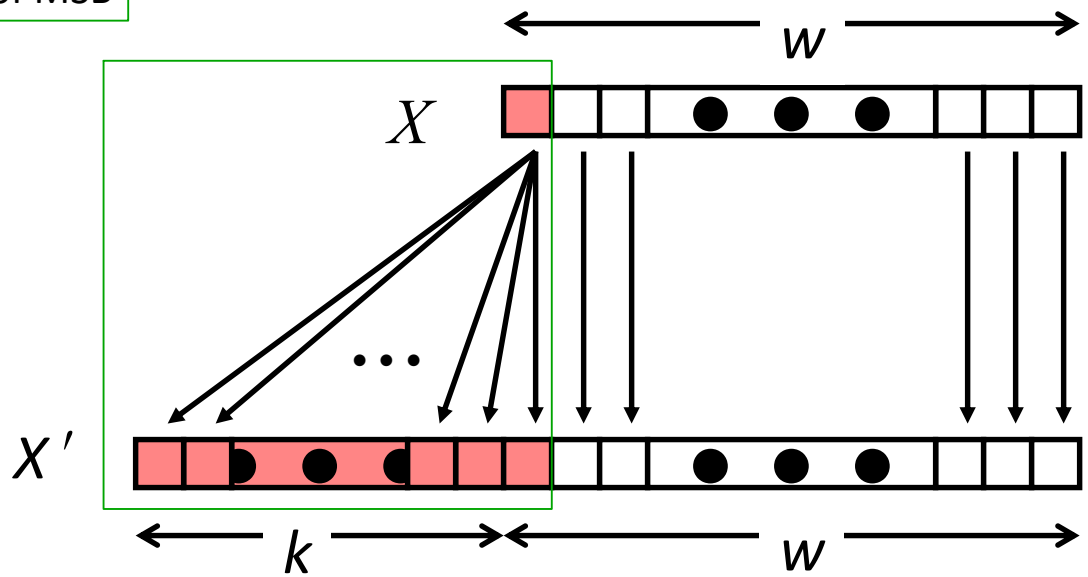
Today: Bits, Bytes, and Integers

- 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:
  - $X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}, x_{w-1}, x_{w-2}, \ldots, x_0$

k copies of MSB



Decimal: -2
Binary (length: 4) 1110
Binary (length: 8) 1111 1110

Sign Extension Example

```
short int x =  15213;
int       ix = (int) x;
short int y = -15213;
int       iy = (int) y;
```

	Decimal	Hex	Binary
x	+15213	3B 6D	00111011 01101101
ix	+15213	00 00 3B 6D	00000000 00000000 00111011 01101101
y	-15213	C4 93	11000100 10010011
iy	-15213	FF FF C4 93	11111111 11111111 11000100 10010011

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

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 나머지 연산!
- Signed: similar to mod
- For small numbers yields expected behavior

Practice

- **Print the result of ‘a’ and ‘b’**

```
int a = 40000 * 40000; // 1600000000
int b = 50000 * 50000; // 2500000000 ??
```

  - Hint: how to print integers

```
printf ("a: %d\n", a);
printf ("b: %d\n", b);
```

- **Compare the following constants.**

Constant$_1$	Constant$_2$	Relation	Evaluation
0	0U	==	unsigned
-1	0	<	signed
-1	0U	>	unsigned

  - Hint: how to define unsigned variables

```
int c1 = 0;
unsigned int c2 = 0U;
```

Today: Bits, Bytes, and Integers

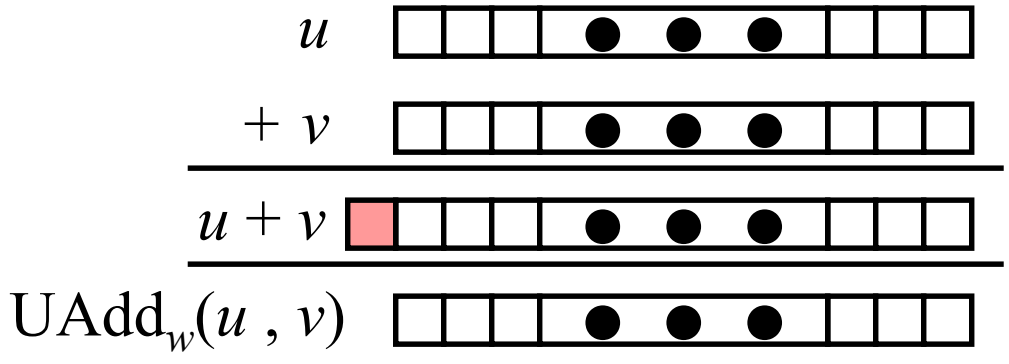
- Representing information as bits
- Bit-level manipulations
- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - **Addition, negation, multiplication, shifting**
- Representations in memory, pointers, strings
- Summary

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



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

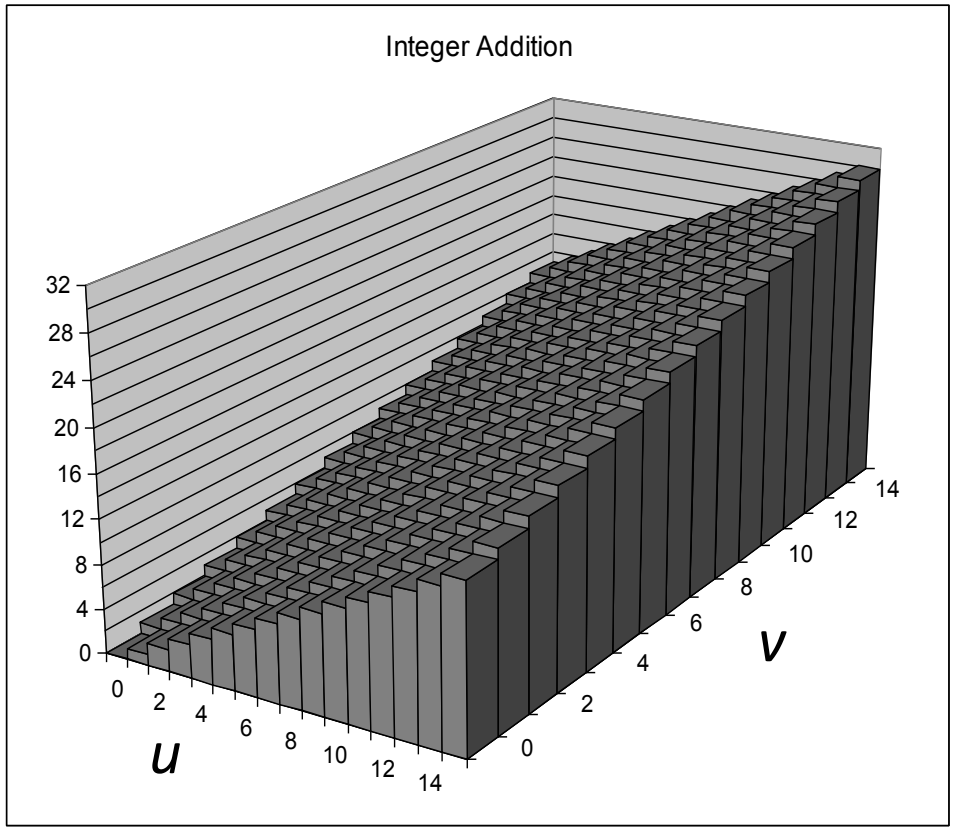
$s = \mathrm{UAdd}_w(u , v) = (u + v) \bmod 2^w$

Visualizing (Mathematical) Integer Addition

Integer Addition

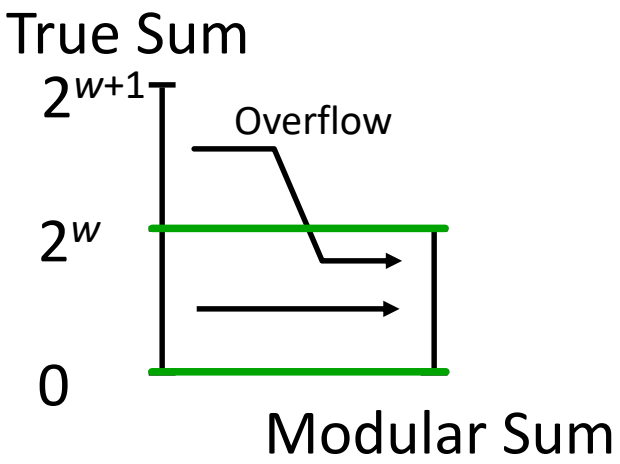
- 4-bit integers *u*, *v*
- Compute true sum $\text{Add}_4(u , v)$
- Values increase linearly with *u* and *v*
- Forms planar surface

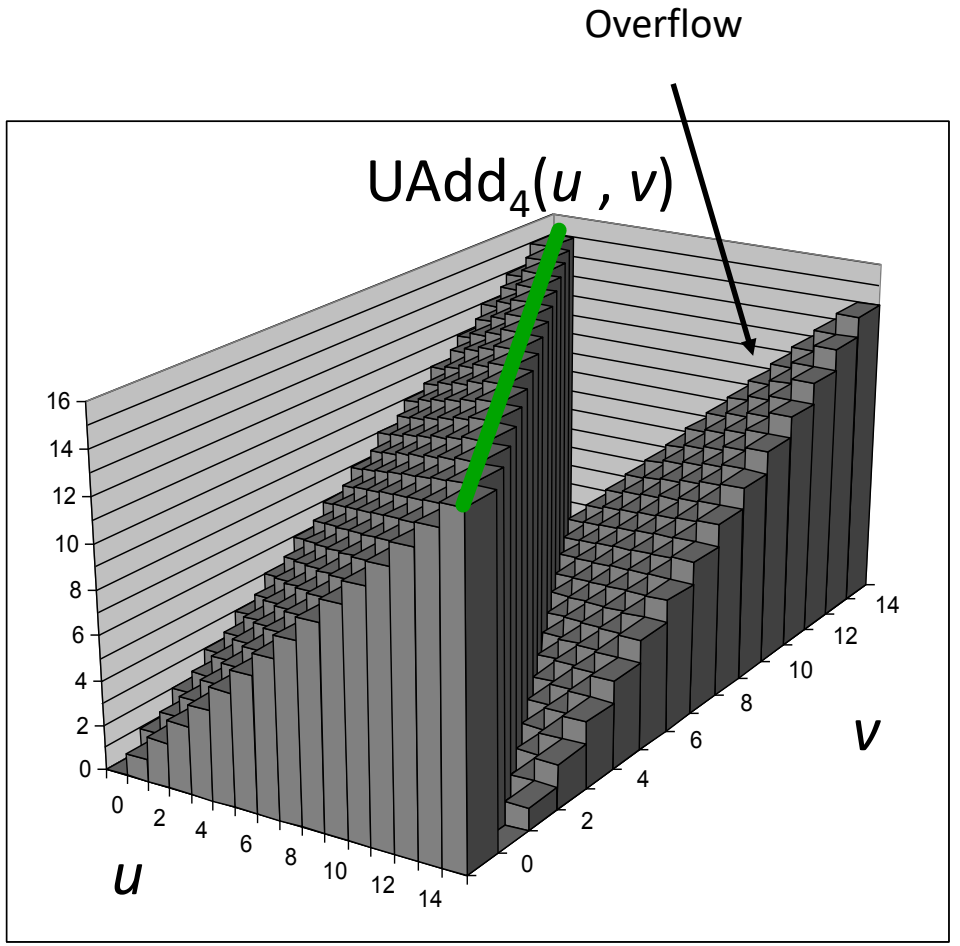
$\text{Add}_4(u , v)$



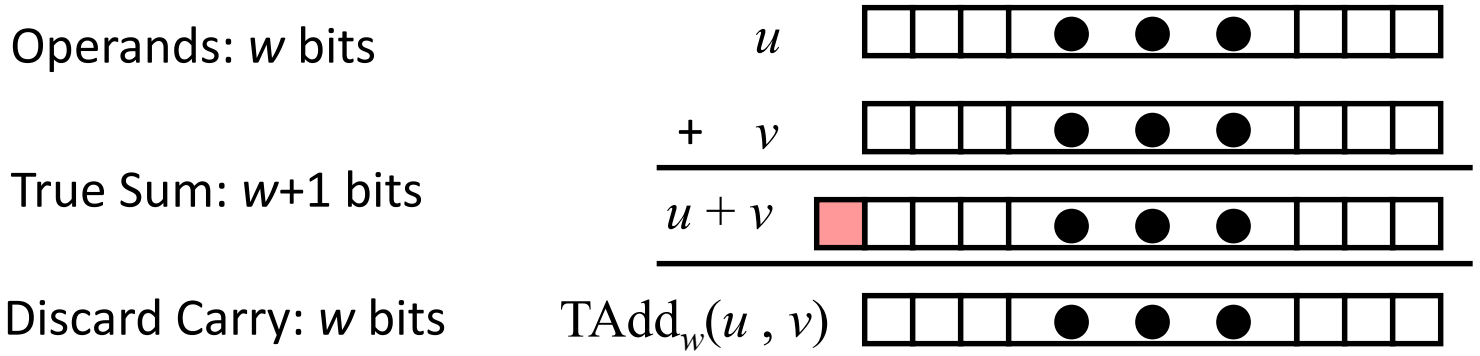
Visualizing Unsigned Addition in Programming

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





Two's Complement Addition



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

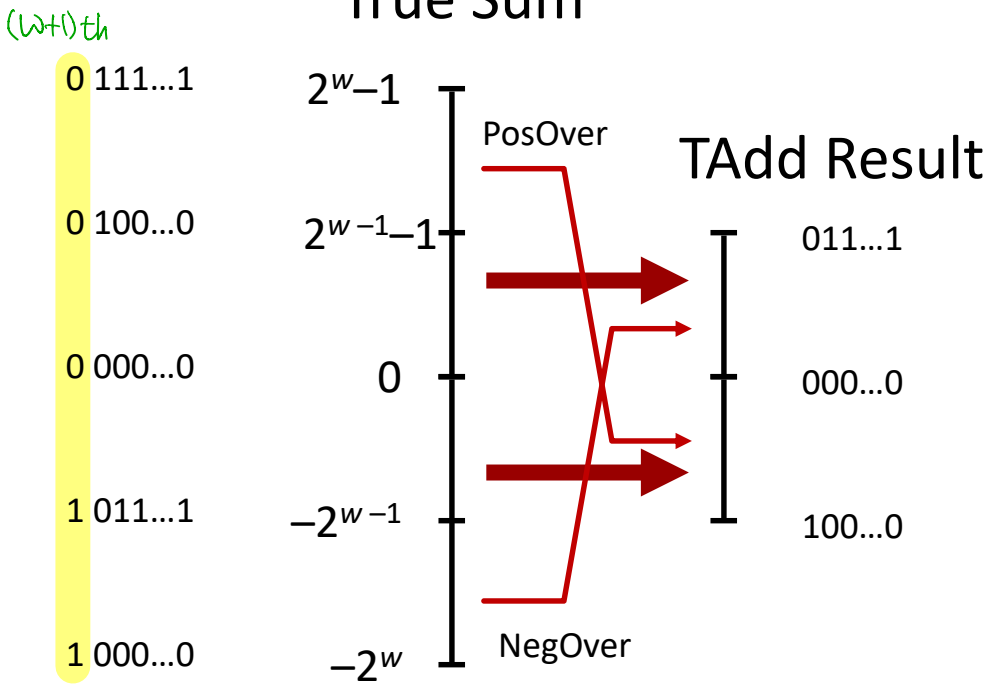
```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

  - Will give `s == t`

TAdd Overflow

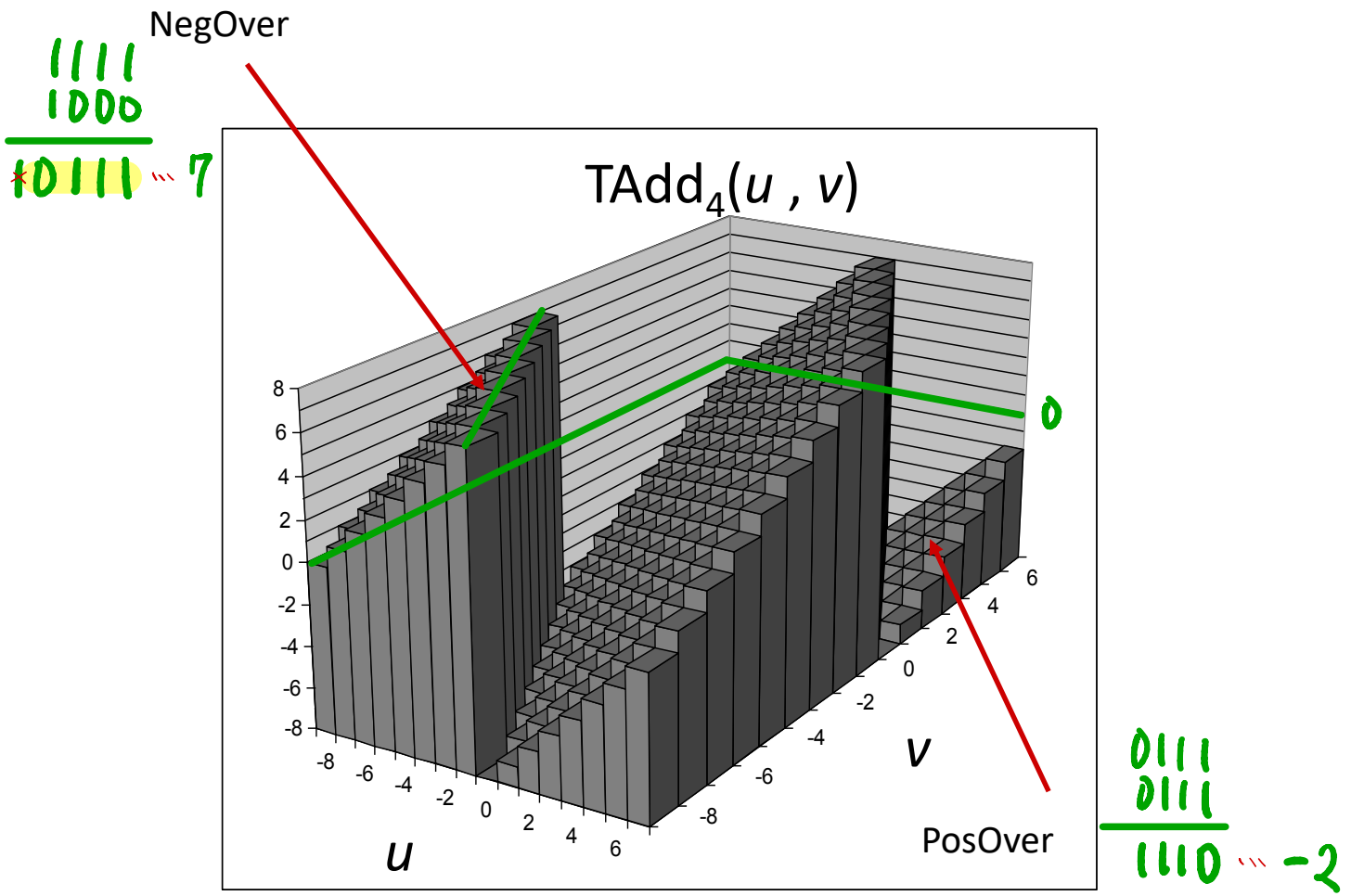
Functionality

- 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 $\geq 2^{w-1}$ PosOver
    - Becomes negative
    - At most once
  - If sum $< -2^{w-1}$ NegOver
    - Becomes positive
    - At most once



Multiplication

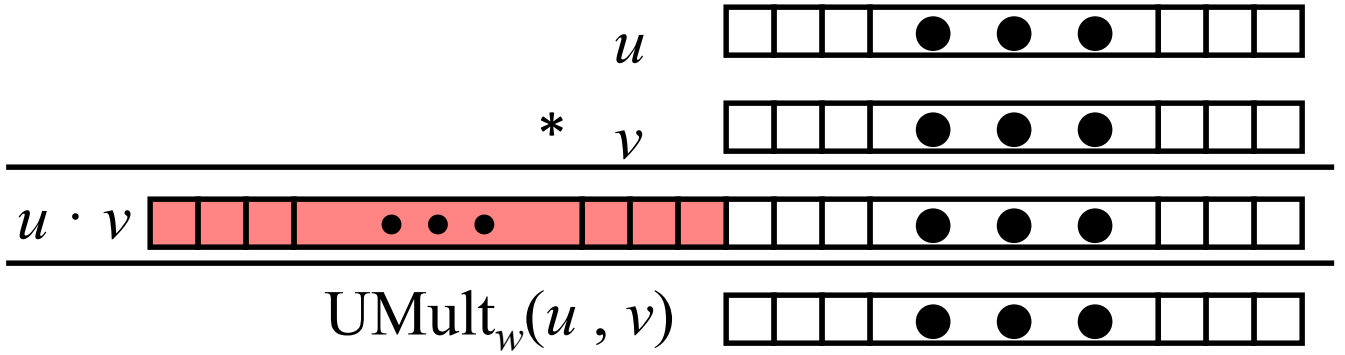
- **Goal: Computing Product of *w*-bit numbers *x*, *y***
  - Either signed or unsigned
- **But, exact results can be bigger than *w* bits**
  - Unsigned: up to 2*w* bits
    - Result range: $0 \le x * y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$
  - Two's complement min (negative): Up to 2*w*-1 bits
    - 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 2*w* 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

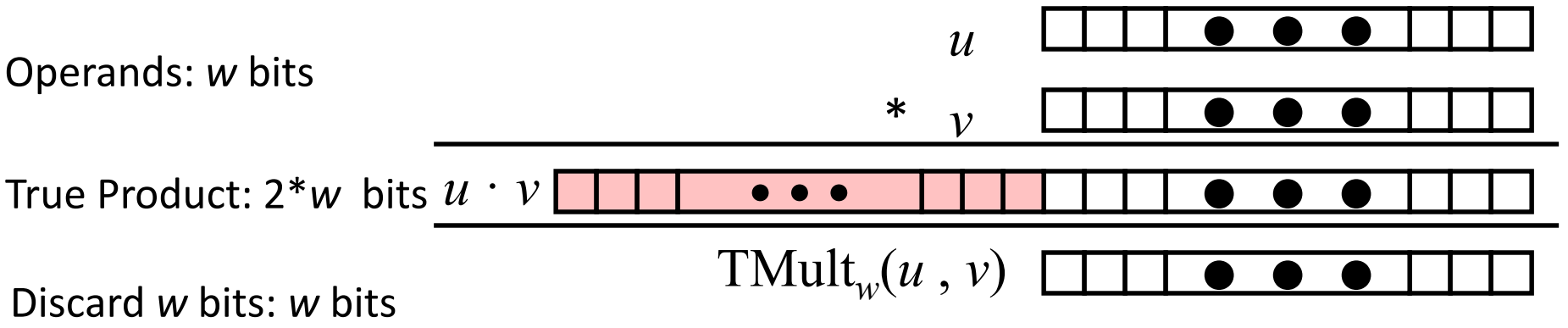
Discard w bits: w bits



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

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

Signed Multiplication in C

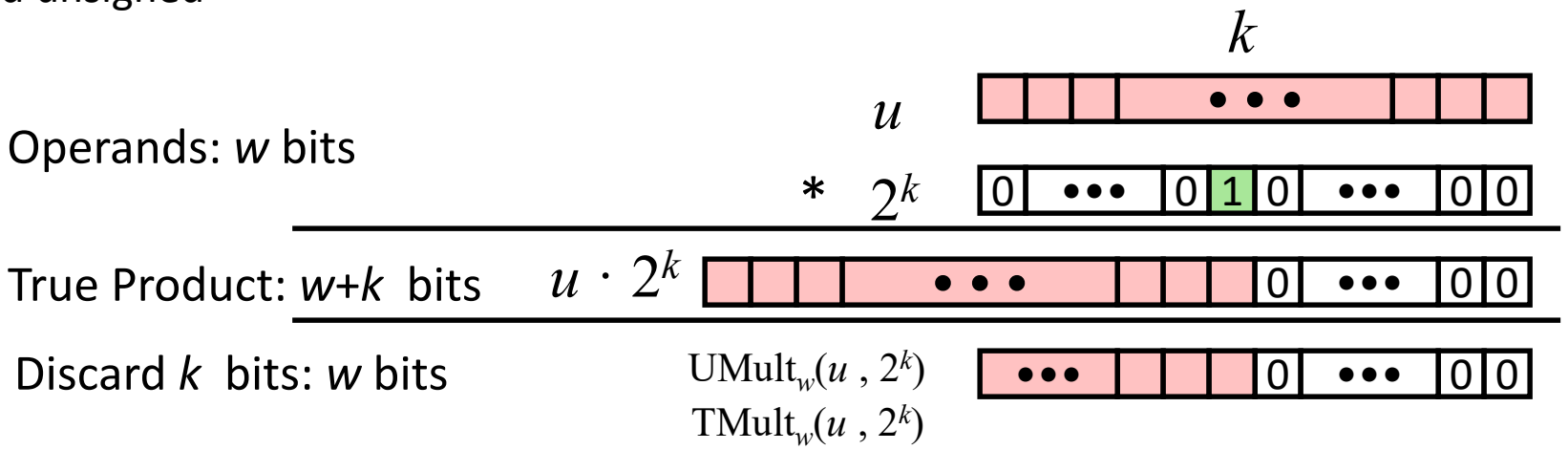


- **Standard Multiplication Function**
  - Ignores high order *w* bits
  - Some of which are different for signed vs. unsigned multiplication

Power-of-2 Multiply with Shift

Operation

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



Examples

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

Q. How can we convert u * 24 into shift operations?

24 − 8 = 24
u<<5 − u<<3

Compiled Multiplication Code

C Function

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

Compiled Arithmetic Operations

```
  leaq (%rax,%rax,2), %rax
  salq $2, %rax
```

Explanation

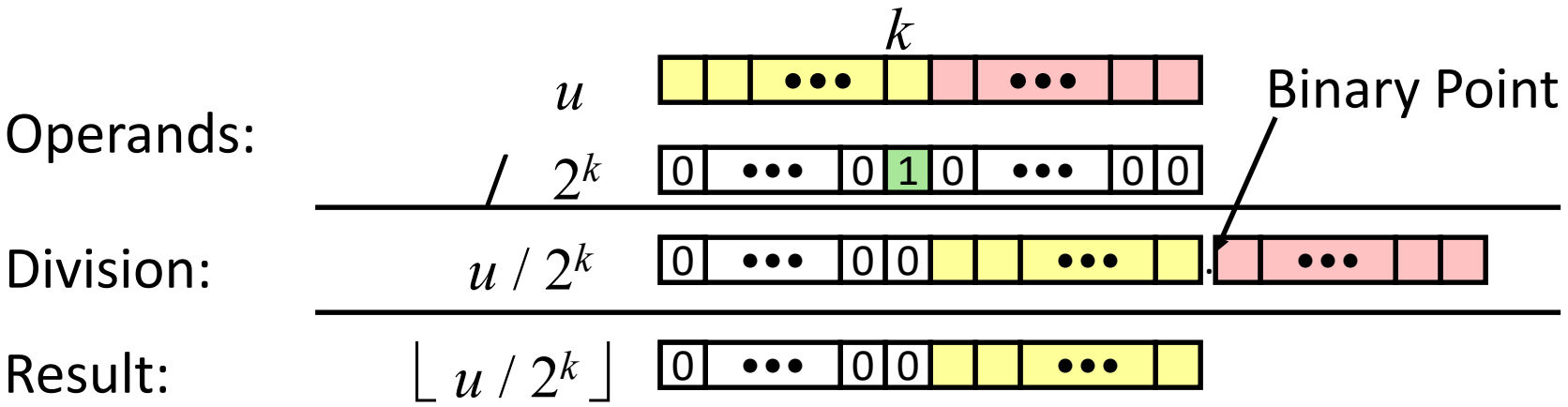
```
  t <- x+x*2
  return t << 2;
```

→ $(x+1) \cdot 2^2$

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

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	1칸
`x >> 4`	950.8125	950	03 B6	00000011 10110110	4칸
`x >> 8`	59.4257813	59	00 3B	00000000 00111011	8칸

Compiled Unsigned Division Code

C Function

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

Compiled Arithmetic Operations

```
  shrq $3, %rax
```

Explanation

```
  # Logical shift
  return x >> 3;
```

- **Uses logical shift for unsigned**

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

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

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

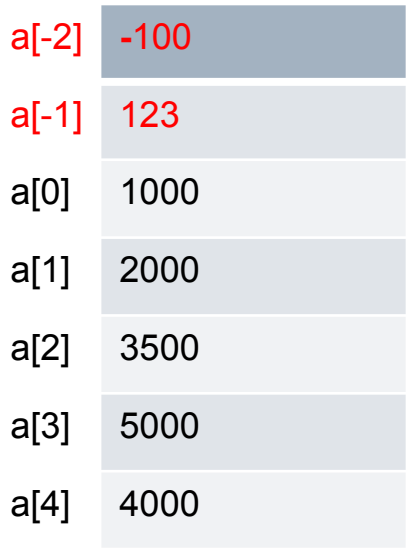
Multiplication:

- 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)

Why Should We Use Unsigned?

- ***Don't* use without understanding implications**
  - Easy to make mistakes

```
int i;
for (i = cnt-2; i >= 0; i--)
  a[i] += a[i+1];
```



Memory layout

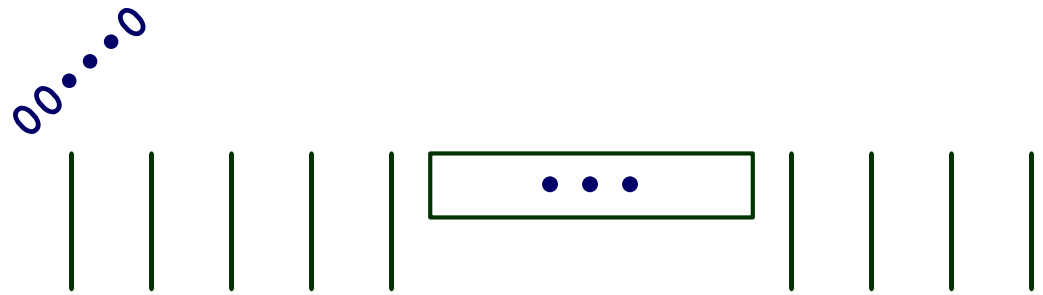
Why Should We Use Unsigned? (cont.)

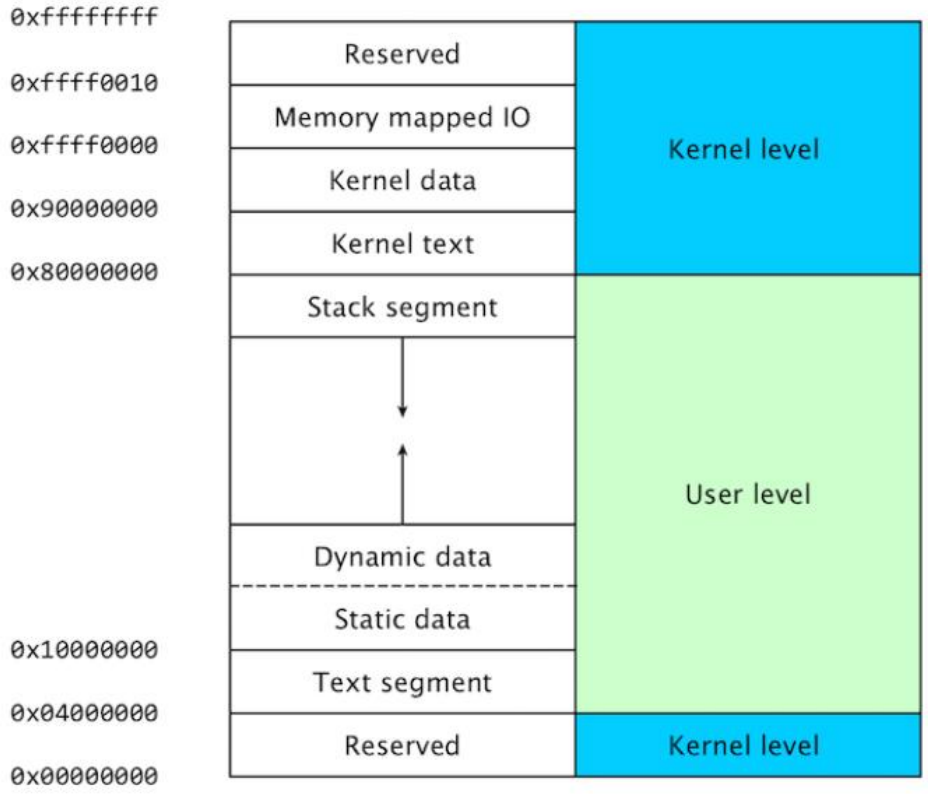
- ***Do*** **Use When Performing Modular Arithmetic**
- ***Do*** **Use When Using Bits to Represent Sets**
  - A = {1, 3, 5, 7} => 01010101

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**

Byte-Oriented Memory Organization





- **Programs refer to data by address**
  - Conceptually, envision it as a very large array of bytes
    - In reality, it's not, but can think of it that way
  - An address is like an index into that array
    - and, a pointer variable stores an address

- **Note: system provides private address spaces to each "process"**
  - Think of a process as a program being executed
  - So, a program can clobber its own data, but not that of others

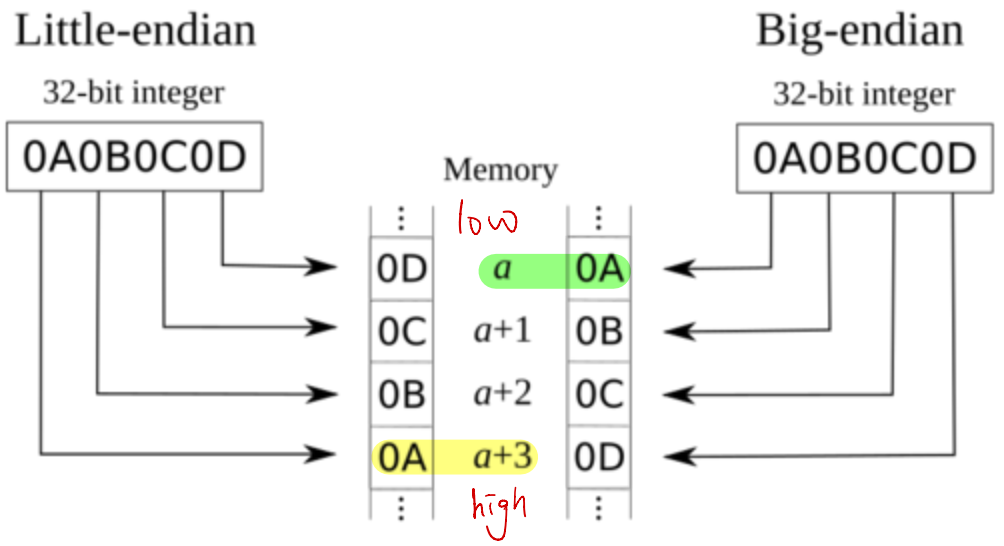
Machine Words

- **Any given computer has a "Word Size"**
  - Nominal size of integer-valued data
    - and of addresses
  - Until recently, most machines used 32 bits (4 bytes) as word size
    - Limits addresses to 4GB (2^{32} bytes)
  - Increasingly, machines have 64-bit word size
    - Potentially, could have 18 EB (exabytes) of addressable memory
    - That's 18.4 X 10^{18}

Byte Ordering

- **So, how are the bytes within a word (multi-bytes) ordered in memory?**

- **Conventions**
  - Big Endian: Sun, PPC Mac, Internet
    - Least significant byte has highest address
  - Little Endian: IA32, X86-64, ARM processors running Android, iOS, and Windows
    - Least significant byte has lowest address



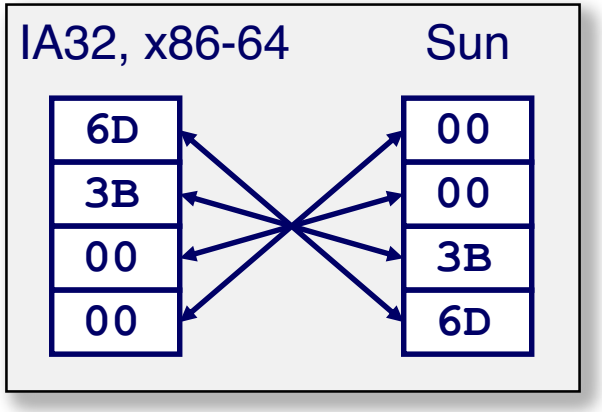
Representing Integers

Decimal:	15213	
Binary:	0011 1011	0110 1101
Hex:	3 B	6 D

1 byte 1 byte

0000 3B 6D

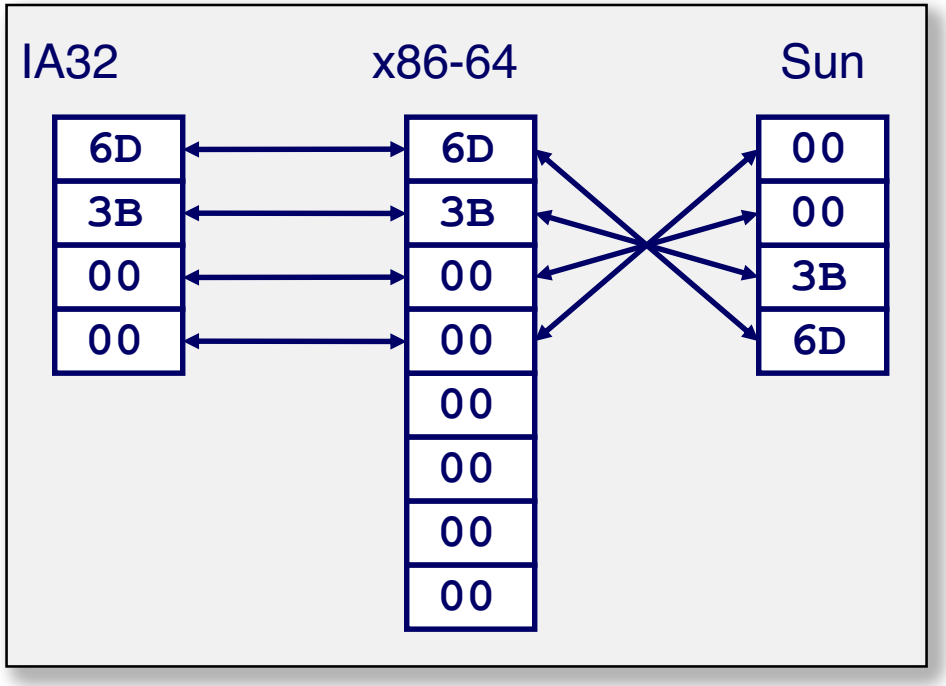
```
int A = 15213;
```



low

high

```
long int C = 15213;
```



low

high

Examining Data Representations

Code to Print Byte Representation of Data

- Casting pointer to unsigned char * allows treatment as a byte array

```
typedef unsigned char *pointer;

void show_bytes(pointer start, size_t len){
  size_t i;
  for (i = 0; i < len; i++)
    printf(”%p %x\n",start+i, start[i]);
  printf("\n");
}
```

Printf directives:
%p: Print pointer
%x: Print Hexadecimal

show_bytes Execution Example

```
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

Little Endian

Result (Linux x86-64):

```
   int a = 15213;
0x7fffb7f71dbc    6d
0x7fffb7f71dbd    3b
0x7fffb7f71dbe    00
0x7fffb7f71dbf    00
```

low

high

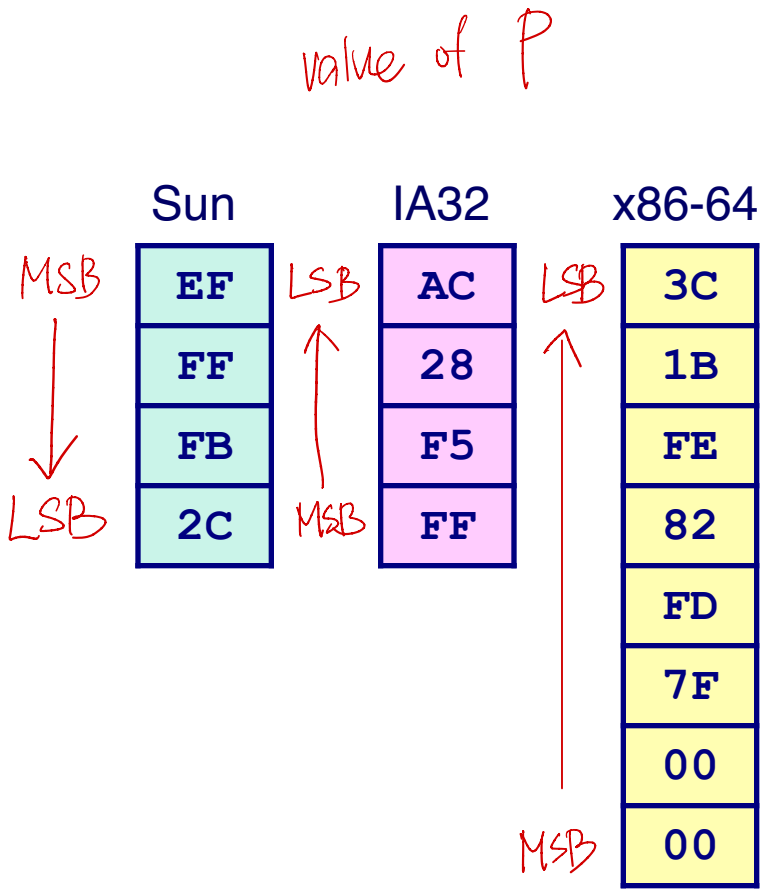
LSB

MSB

15213 → 00 00 3b 6d

Representing Pointers

```
int B = -15213;
int *P = &B;
```



Different compilers & machines assign different locations to objects

Even get different results each time run program

Representing Strings

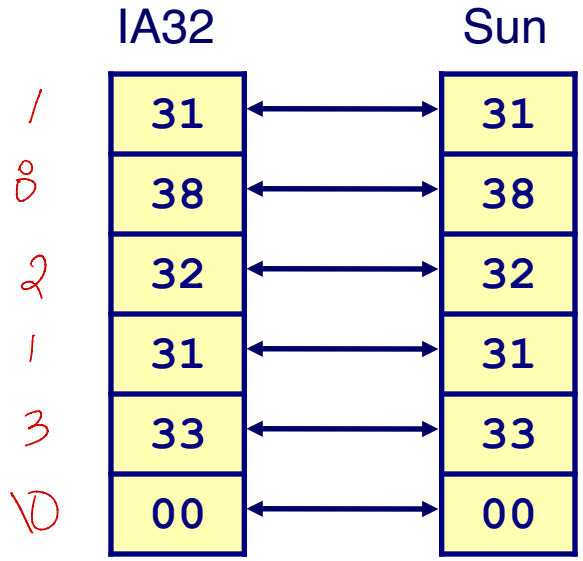
Strings in C

- Represented by array of characters
- Each character encoded in ASCII format
  - Standard 7-bit encoding of character set
  - Character “0” has code 0x30
    - Digit *i* has code 0x30+*i*
- String should be null-terminated
  - Final character = 0

Compatibility

- Byte ordering not an issue

```
char S[6] = "18213";
```



Practice

- **Check the actual byte ordering method on your environment**

```
typedef unsigned char *pointer;

void show_bytes(pointer start, size_t len){
  size_t i;
  for (i = 0; i < len; i++)
    printf(”%p %x\n",start+i, start[i]);
  printf("\n");
}
```

```
int a = 15213;
printf("int a = 15213;\n");
show_bytes((pointer) &a, sizeof(int));
```

Today: Floating Point

- Background: Fractional binary numbers
- IEEE floating point standard: Definition
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

Fractional binary numbers

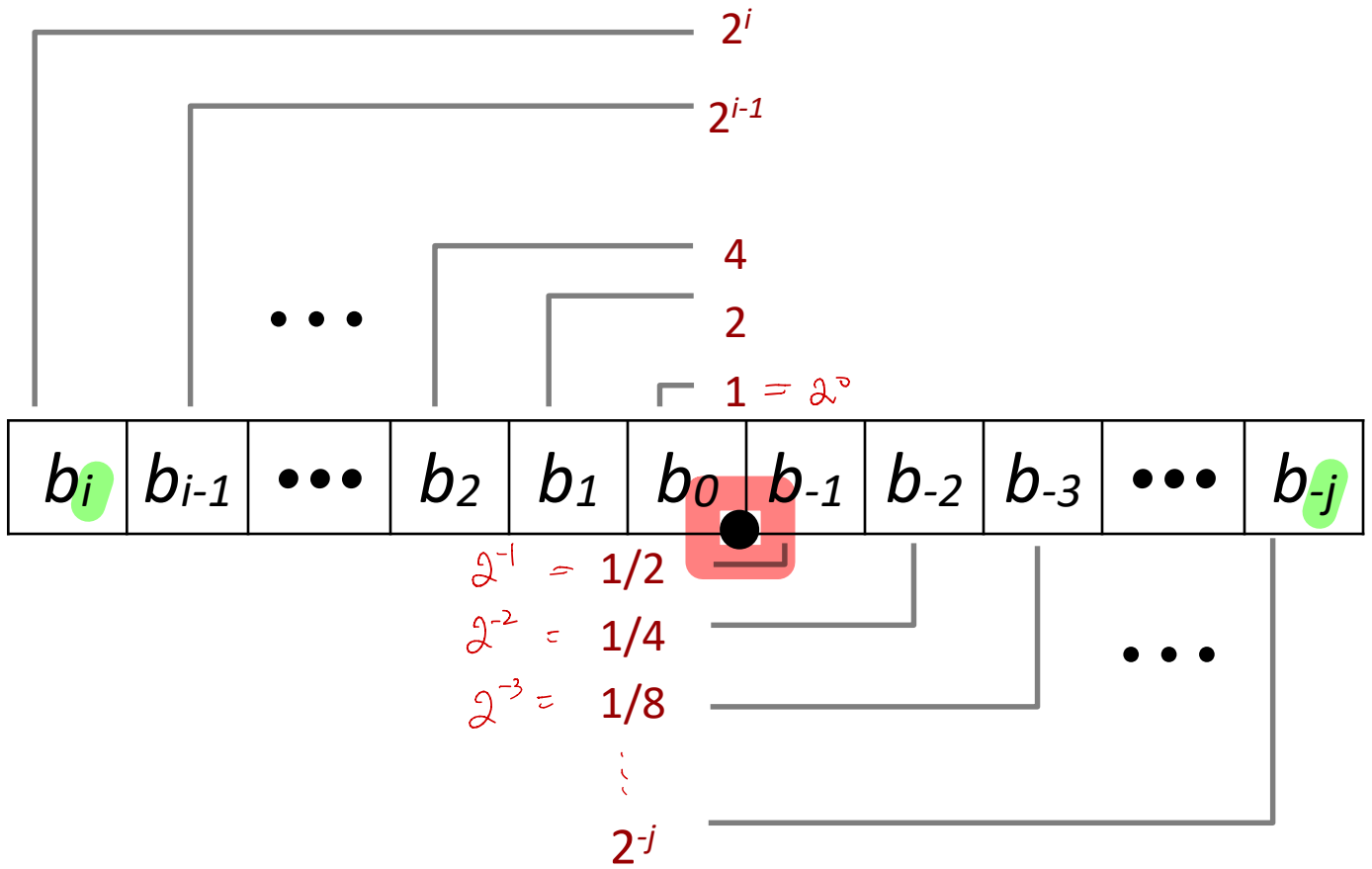
- What is 1011.101_2?

2^3 2^2 2^1 2^0 2^{-1} 2^{-2} 2^{-3}

1 0 1 1 1 0 1

$= 8+2+1+\frac{1}{2}+\frac{1}{8} = 11+\frac{5}{8} = 11.625$

Fractional Binary Numbers



Representation

- Bits to right of "binary point" represent fractional powers of 2
- Represents rational number:

$$\sum_{k=-j}^{i} b_k \times 2^k$$

Fractional Binary Numbers: Examples

Value	Representation
5 3/4	Q1 101.11
2 7/8	Q2 10.111
1 7/16	Q3 1.0111

- **Observations**
  - Divide by 2 by shifting right (unsigned) — logical shift
  - Multiply by 2 by shifting left
  - Numbers of form $0.111111\ldots_2$ are just below 1.0

Representable Numbers

Limitation #1

- Can only exactly represent numbers of the form $x/2^k$
  - Other rational numbers have repeating bit representations

Value	Representation
1/3	$0.0101010101[01]\ldots_2$
1/5	$0.001100110011[0011]\ldots_2$
1/10	$0.0001100110011[0011]\ldots_2$

Limitation #2

- Just one setting of binary point within the *w* bits
  - Limited range of numbers (very small values? very large?)

Today: Floating Point

- Background: Fractional binary numbers
- **IEEE floating point standard: Definition**
- Example and properties
- Rounding, addition, multiplication
- Floating point in C
- Summary

IEEE Floating Point

IEEE Standard 754

- Established in 1985 as uniform standard for floating point arithmetic
  - Before that, many idiosyncratic formats
- Supported by all major CPUs

Floating Point Representation

Numerical Form:

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

- **Sign bit *s*** determines whether number is negative or positive
- **Significand *M*** normally a fractional value in range [1.0,2.0).
- **Exponent *E*** weights value by power of two

$(+1) \times (1.1101_2) \times (2^{13})$

↳ $(1.0 \le M < 2.0)$

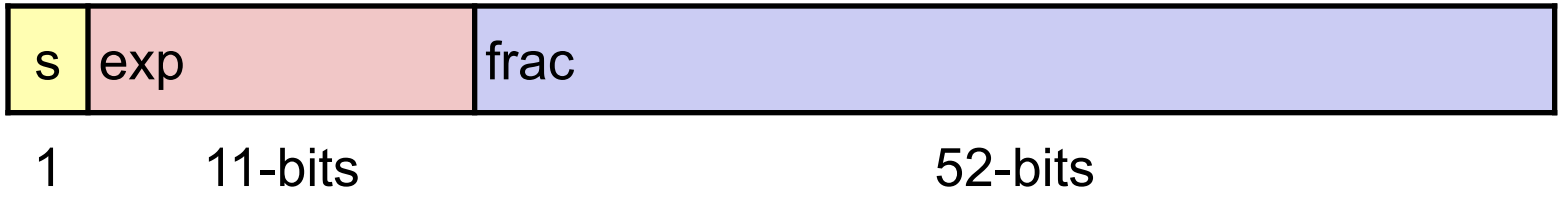
Encoding

- MSB s is sign bit ***s***
- exp field encodes ***E*** (but is not equal to E)
- frac field encodes ***M*** (but is not equal to M)

Precision options

- **Single precision: 32 bits (float)**

- **Double precision: 64 bits (double)**



- **Extended precision: 80 bits (Intel only- long double)**

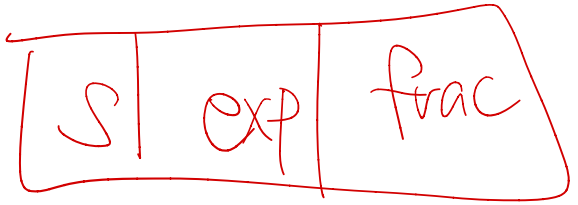
accuracy ↑

"Normalized" Values

s exp frac.
1 8 23
11 52

- **Exponent coded as a *biased* value: *E* = *Exp* – *Bias***
  - *Exp*: unsigned value of exp field
  - *Bias* = $2^{k-1} - 1$, where *k* is number of exponent bits
    - Single precision: 127 (Exp: 1…254, E: -126…127)
    - Double precision: 1023 (Exp: 1…2046, E: -1022…1023)

$v = (-1)^s M 2^E$



32-bits bias = $2^{8-1}-1 = 127$

64-bits bias = $2^{11-1}-1 = 1023$

range of E : $-bias+1 \le E \le bias$

- **Significand coded with implied leading 1: *M* = 1.xxx…x$_2$**
  - xxx…x: bits of frac field
  - Minimum when frac=000…0 (M = 1.0)
  - Maximum when frac=111…1 (M = 2.0 – ε)

frac

1.0000…0 = 1

1.1111…1 < 2.0

(very close to 2.0, but under 2.0)

임의의 아주작은 양수

Normalized Encoding Example

- **Value:** `float F = 15213.0;`
  - $15213_{10} = 11101101101101_2$
    $= 1.1101101101101_2 \times 2^{13}$

4byte = 32bits ⇒ exp 8bits frac 23bits

- **Significand**

$M = 1.\underline{1101101101101}_2$

frac= $\underline{1101101101101}0000000000_2$ 23bits

frac

- **Exponent**

$E = 13$ = Exp − Bias

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

$Exp = 140 = 10001100_2$

- **Result:**

0	10001100	11011011011010000000000
s	exp = 140	frac

$(-1)^0$ bias = 127 1.1101101101101_2 → $(-1)^0 \times 1.1101101101101_2 \times 2^{140-127} = 1.1101101101101_2 \times 2^{13}$

$= 1110110110110_2 = 15213$

37541

$10010010\ 01001010_2 = 1.00100101\ 01001010_2 \times 2^{15}$

s = 0

exp = 15+127 = 142

frac = 00100101010010100…0

$$v = (-1)^s M\, 2^E$$
$$E = Exp - Bias$$

Denormalized Values

- **Condition: exp = 000…0**

  In denormalized encoding,

  $E = 1 - bias$.

  32-bits | $E = 1 - (2^{n-1} - 1) = -126$

  64-bits | $E = 1 - (2^{n-1} - 1) = -1022$

$$v = (-1)^s M 2^E$$

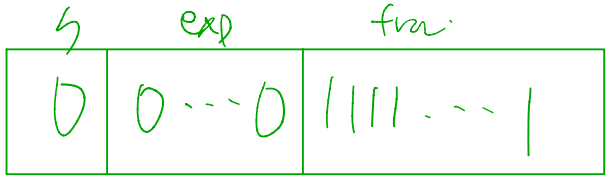
- **Significand coded with implied leading 0: *M* = 0.xxx…x_2**
  - **xxx**…**x**: bits of **frac**

  if $frac = 1100_2$

  $exp = 0_2$

  $s = 0$

- **Cases**
  - **exp** = 000…0, **frac** = 000…0
    - Represents zero value
  - **exp** = 000…0, **frac** ≠ 000…0
    - Numbers closest to 0.0



n-bit bias = 127

$E = 1 - bias$

$frac = 0b(M)$

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

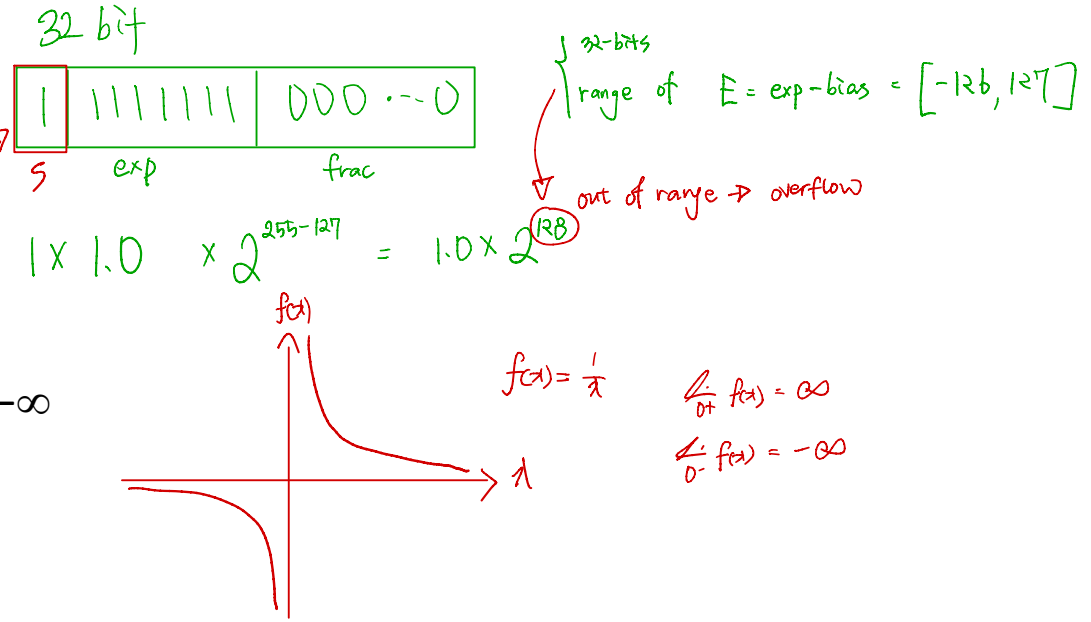
$0 < 1 \times 0.11\ldots1_2 \times 2^{-126}$

but close to 0.0

Special Values

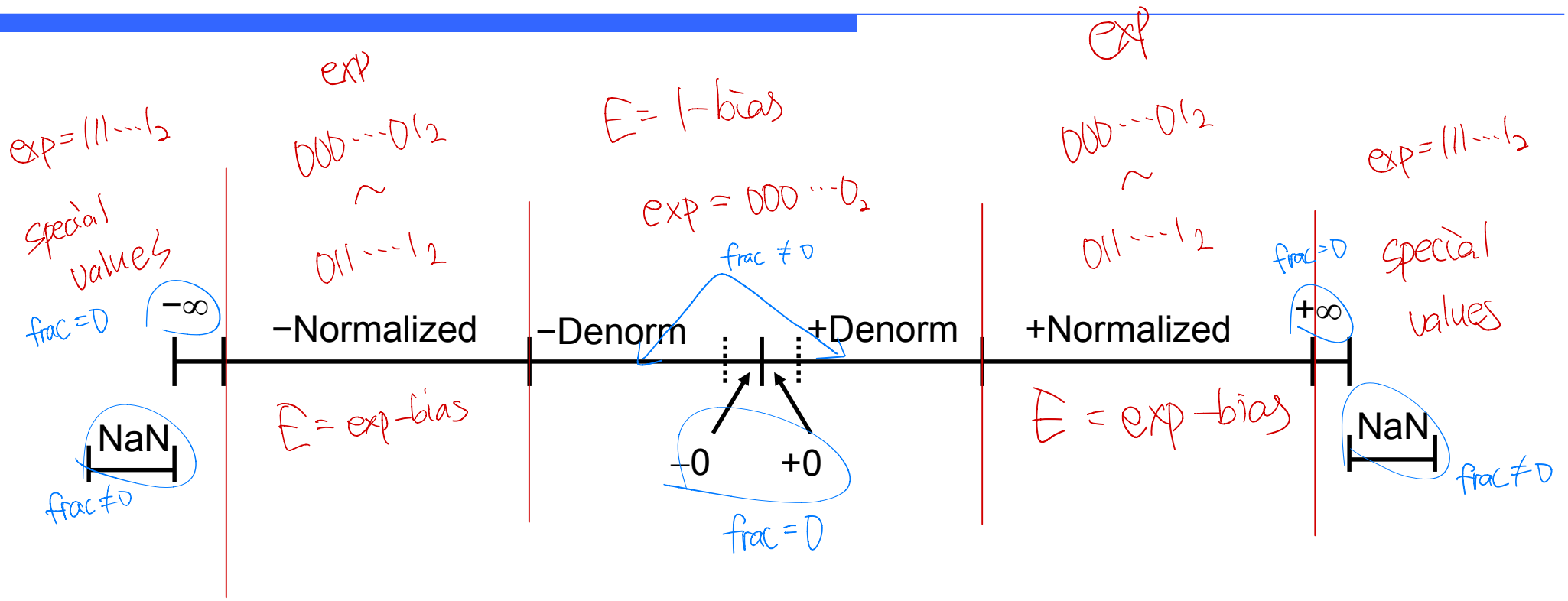
- **Condition: `exp` = `111…1`**

- **Case: `exp` = `111…1`, `frac` = `000…0`**
  - Represents value ∞ (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., 1.0/0.0 = −1.0/−0.0 = +∞, 1.0/−0.0 = −∞



- **Case: `exp` = `111…1`, `frac` ≠ `000…0`**
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), ∞ − ∞, ∞ × 0

Visualization: Floating Point Encodings

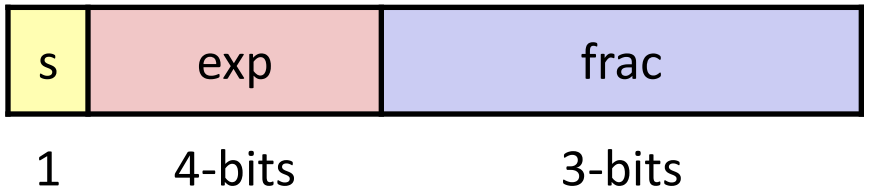


Practice: Creating Floating Point Number

$$v = (-1)^s M 2^E$$
$$E = Exp - Bias$$

Requirement

- Set binary point so that numbers of form 1.xxxxx
- Adjust all to have leading one
  - Decrement exponent as shift left



bias = $2^3 - 1 = 7$

Value	*Binary*	*Fraction*	*E (Exponent)*
128	10000000	1.0000000	7
15	0000 1101	1.1010000	3
17	00010001	1.0001000	4
138	10001010	1.0001010	7
63	00111111	1.1111100	5

$-6 \le E (Exponent) \le 7$

exp = 0 | $E = 1 - 7 = -6$

exp = 15 | ∞ or NaN

$0 < exp < 15$ | exp - bias = E

$-6 \le E \le 7$