CSC 252/452: Computer Organization
Fall 2024: Lecture 3

Instructor: Yanan Guo
Department of Computer Science
University of Rochester

Announcement

- Programming Assignment 1 is out
  - Details: https://www.cs.rochester.edu/courses/252/fall2024/labs/assignment1.html
  - Due on Sep 16th, 11:59 PM
  - You have 3 slip days

Announcement

- Programming Assignment 1 is in C language.
- Seek help from TAs.
  - TAs are best positioned to answer your questions about programming assignments!!!
- Programming assignments do NOT repeat the lecture materials. They ask you to synthesize what you have learned from the lectures and work out something new.
- Pay attention to Blackboard announcements
  - There are changes about the office hour locations/time…
  - I have to move my office hour tomorrow to early next week.

Last Lecture

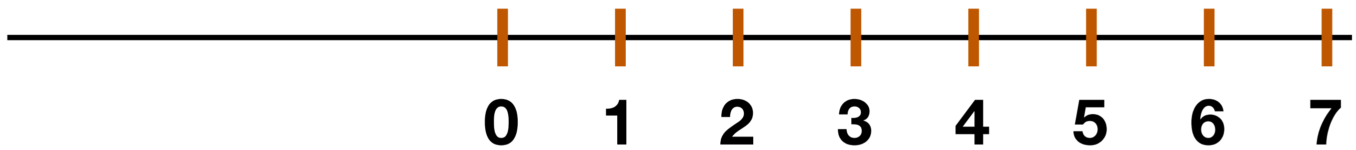
- Why Binary (bits)?
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary

Encoding Negative Numbers

- Two’s Complement

Encoding Negative Numbers

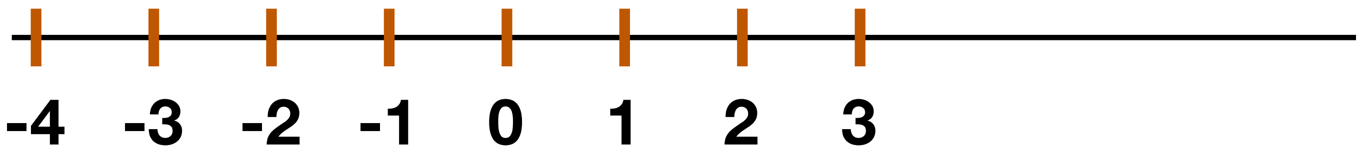
- Two's Complement



Unsigned	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Encoding Negative Numbers

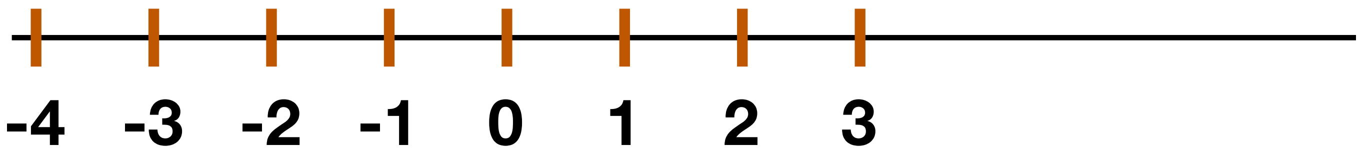
- Two's Complement



Unsigned	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Encoding Negative Numbers

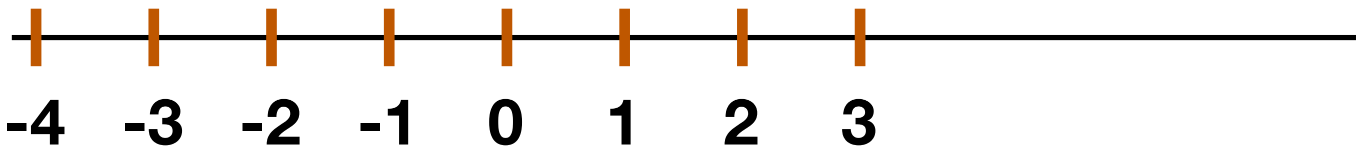
- Two’s Complement



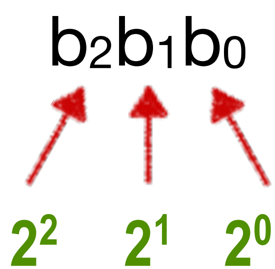
Signed	Unsigned	Binary
0	0	000
1	1	001
2	2	010
3	3	011
-4	4	100
-3	5	101
-2	6	110
-1	7	111

Encoding Negative Numbers

- Two's Complement



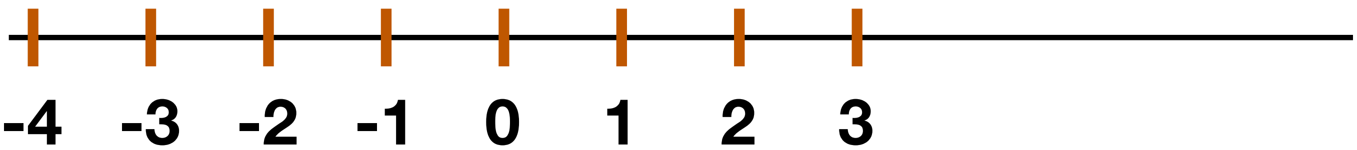
Weights in Unsigned



Signed	Unsigned	Binary
0	0	000
1	1	001
2	2	010
3	3	011
-4	4	100
-3	5	101
-2	6	110
-1	7	111

Encoding Negative Numbers

- Two's Complement



$b_2 b_1 b_0$

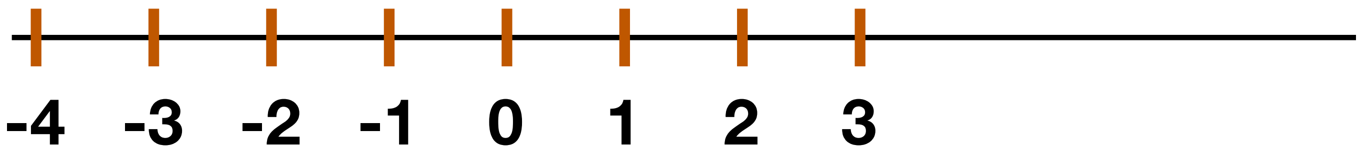
Weights in Unsigned: 2^2 2^1 2^0

Weights in Signed: -2^2 2^1 2^0

Signed	Unsigned	Binary
0	0	000
1	1	001
2	2	010
3	3	011
-4	4	100
-3	5	101
-2	6	110
-1	7	111

Encoding Negative Numbers

- Two’s Complement



$b_2 b_1 b_0$

Weights in Unsigned: 2^2 2^1 2^0

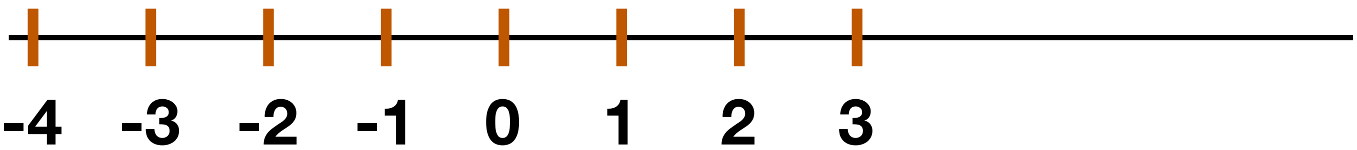
Weights in Signed: -2^2 2^1 2^0

Signed	Unsigned	Binary
0	0	000
1	1	001
2	2	010
3	3	011
-4	4	100
-3	5	101
-2	6	110
-1	7	111

$$101_2 = 1*2^0 + 0*2^1 + (-1*2^2) = -3_{10}$$

Encoding Negative Numbers

- Two's Complement



$b_2 b_1 b_0$

Weights in Unsigned: $2^2 \quad 2^1 \quad 2^0$

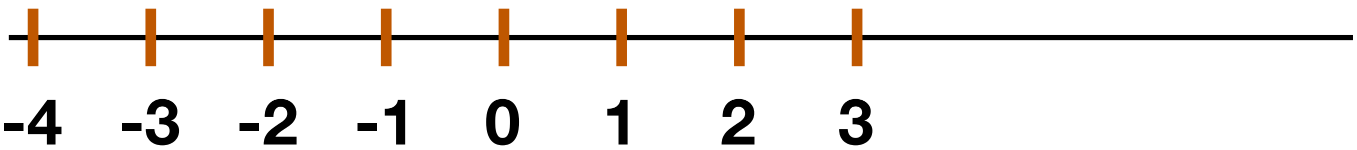
Weights in Signed: $-2^2 \quad 2^1 \quad 2^0$

$$101_2 = \boxed{1*2^0} + 0*2^1 + (-1*2^2) = -3_{10}$$

Signed	Unsigned	Binary
0	0	000
1	1	001
2	2	010
3	3	011
-4	4	100
-3	5	101
-2	6	110
-1	7	111

Encoding Negative Numbers

- Two's Complement



$b_2 b_1 b_0$

Weights in Unsigned 2^2 2^1 2^0

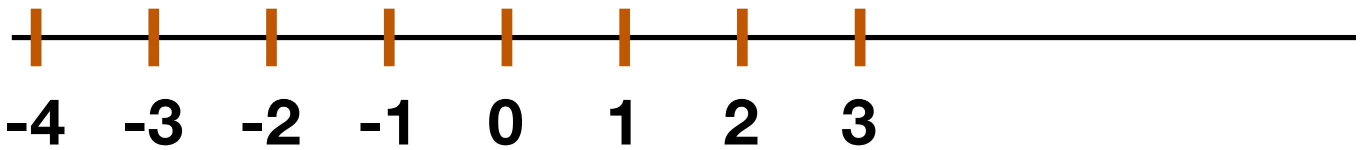
Weights in Signed -2^2 2^1 2^0

$101_2 = \boxed{1*2^0} + 0*2^1 + (-1*2^2) = -3_{10}$

Signed	Unsigned	Binary
0	0	000
1	1	001
2	2	010
3	3	011
-4	4	100
-3	5	101
-2	6	110
-1	7	111

Encoding Negative Numbers

- Two’s Complement



$b_2 b_1 b_0$

Weights in Unsigned 2^2 2^1 2^0

Weights in Signed -2^2 2^1 2^0

$$101_2 = \boxed{1*2^0} + 0*2^1 + (-1*2^2) = -3_{10}$$

Signed	Unsigned	Binary
0	0	000
1	1	001
2	2	010
3	3	011
-4	4	100
-3	5	101
-2	6	110
-1	7	111

Two-Complement Implications

- Only 1 zero
- There is (still) a bit that represents sign!
- Unsigned arithmetic still works

Signed	Binary
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

Two-Complement Implications

- Only 1 zero
- There is (still) a bit that represents sign!
- Unsigned arithmetic still works

```
    010
+)  101
-------
    111
```

Signed	Binary
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

Two-Complement Implications

- Only 1 zero
- There is (still) a bit that represents sign!
- Unsigned arithmetic still works

```
    010          2
+)  101      +) -3
-------      -----
    111         -1
```

Signed	Binary
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

Two-Complement Implications

- Only 1 zero
- There is (still) a bit that represents sign!
- Unsigned arithmetic still works

Signed	Binary
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

```
    010
+)  101
-------
    111
```

```
      2
+)   -3
-------
     -1
```

- 3 + 1 becomes -4 (called overflow. More on it later.)

Data Types (in C)

- Suppose you want to define a variable that represents a person's age. What assumptions can you make about this variable's numerical value?

Data Types (in C)

- Suppose you want to define a variable that represents a person's age. What assumptions can you make about this variable's numerical value?
  - Integer
  - Non-negative
  - Between 0 and 255 (8 bits)

Data Types (in C)

- Suppose you want to define a variable that represents a person's age. What assumptions can you make about this variable's numerical value?
  - Integer
  - Non-negative
  - Between 0 and 255 (8 bits)
- Define a data type that captures all these attributes: `unsigned char` in C
  - Internally, an **`unsigned char`** variable is represented as a 8-bit, non-negative, binary number

Data Types (in C)

- What if you want to define a variable that could take negative values?

Data Types (in C)

- What if you want to define a variable that could take negative values?
    - That's what signed data types (e.g., **`int`**, **`short`**, etc.) are for

Data Types (in C)

- What if you want to define a variable that could take negative values?
  - That's what signed data types (e.g., **`int`**, **`short`**, etc.) are for
- How are `int` values internally represented?
  - Theoretically could be either signed-magnitude or two's complement
  - The C language designers chose two's complement

Data Types (in C)

	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

C Data Type	32-bit	64-bit
`(unsigned) char`	1	1
`(unsigned) short`	2	2
`(unsigned) int`	4	4
`(unsigned) long`	4	8

Data Types (in C)

	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

C Data Type	32-bit	64-bit
`(unsigned) char`	1	1
`(unsigned) short`	2	2
`(unsigned) int`	4	4
`(unsigned) long`	4	8

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

Mapping Between Signed & Unsigned

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

Signed	Unsigned	Binary
0	0	000
1	1	001
2	2	010
3	3	011
-4	4	100
-3	5	101
-2	6	110
-1	7	111

Mapping Signed ↔ Unsigned

Bits	Signed		Unsigned
0000	0		0
0001	1		1
0010	2		2
0011	3	=	3
0100	4		4
0101	5	T2U	5
0110	6		6
0111	7		7
1000	-8	U2T	8
1001	-7		9
1010	-6	+/- 16	10
1011	-5		11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

Today: Representing Information in Binary

- Why Binary (bits)?
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary

The Problem

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

C Data Type	64-bit
char	1
short	2
int	4
long	8

The Problem

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

C Data Type	64-bit
`char`	1
`short`	2
`int`	4
`long`	8

- Converting from smaller to larger integer data type
- Should we preserve the value?
- Can we preserve the value?
- How?

The Problem

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

C Data Type	64-bit
`char`	1
`short`	2
`int`	4
`long`	8

- Converting from smaller to larger integer data type
- Should we preserve the value?
- Can we preserve the value?
- How?

	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

Signed Extension

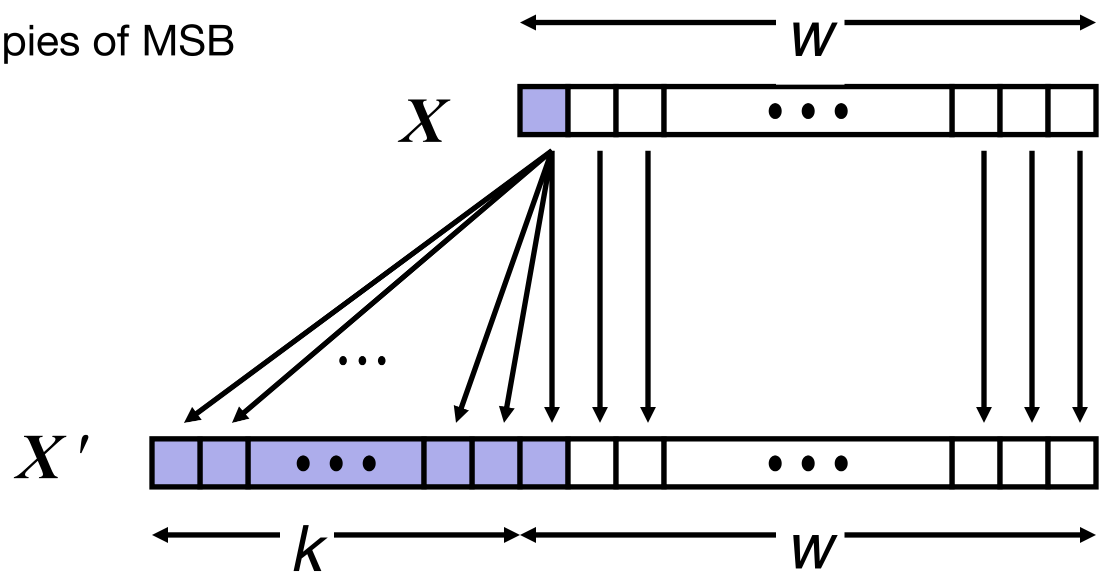
- Task:
  - Given *w*-bit signed integer *x*
  - Convert it to (*w*+*k*)-bit integer with same value

Signed 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:
  - $X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \ldots, x_0$

Signed 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:
  - $X' = \underbrace{x_{w-1}, \ldots, x_{w-1}}_{k \text{ copies of MSB}}, x_{w-1}, x_{w-2}, \ldots, x_0$



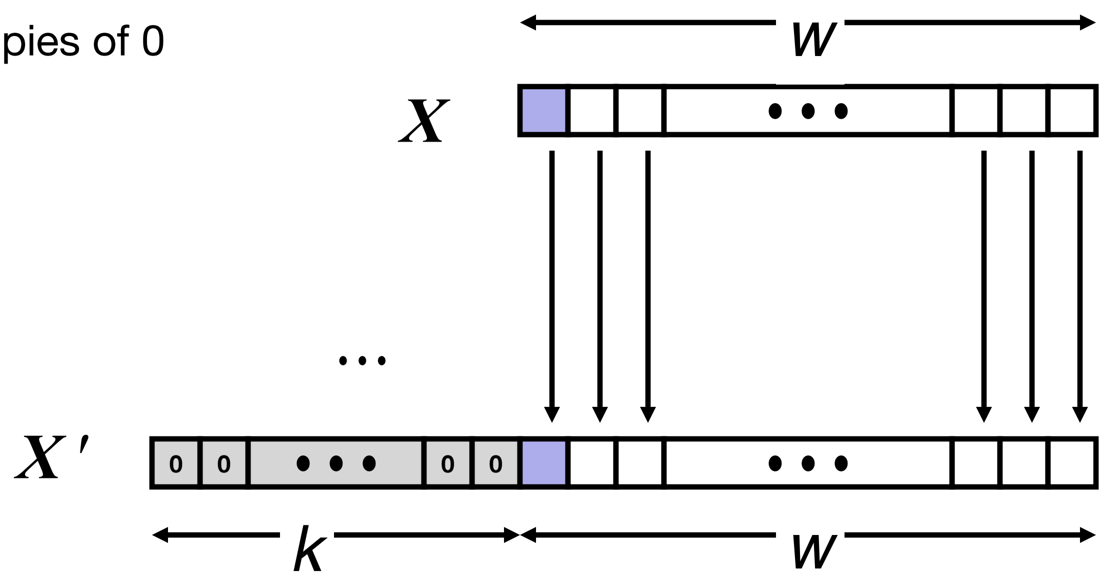
Another Problem

```
unsigned short x = 47981;
unsigned int  ux = x;
```

	Decimal	Hex	Binary
x	47981	BB 6D	10111011 01101101
ux	47981	00 00 BB 6D	00000000 00000000 10111011 01101101

Unsigned (Zero) Extension

- Task:
  - Given w-bit unsigned integer x
  - Convert it to $(w+k)$-bit integer with same value
- Rule:
  - Simply pad zeros:
  - $X' = \underbrace{0, \ldots, 0}_{k \text{ copies of } 0}, x_{w-1}, x_{w-2}, \ldots, x_0$



Yet Another Problem

```
int    x =  53191;
short sx = (short) x;
```

	Decimal	Hex	Binary
x	53191	00 00 CF C7	00000000 00000000 11001111 11000111
sx	-12345	CF C7	11001111 11000111

Yet Another Problem

```
int     x =  53191;
short sx = (short) x;
```

	Decimal	Hex	Binary
x	53191	00 00 CF C7	00000000 00000000 11001111 11000111
sx	-12345	CF C7	11001111 11000111

- Truncating (e.g., int to short OR unsigned int to unsigned short)
  - C's implementation: leading bits are truncated, results reinterpreted
  - So can't always preserve the numerical value

Today: Representing Information in Binary

- Why Binary (bits)?
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary
- Representations in memory, pointers, strings

Unsigned Addition

Unsigned	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Unsigned Addition

- Similar to Decimal Addition

Unsigned	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Unsigned Addition

- Similar to Decimal Addition
- Suppose we have a new data type that is 3-bit wide (c.f., **`short`** has 16 bits)

Unsigned	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Normal Case

```
    010          2
+)  101     +)   5
-------     ------
    111          7
```

Unsigned Addition

- Similar to Decimal Addition
- Suppose we have a new data type that is 3-bit wide (c.f., **`short`** has 16 bits)
- Might overflow: result can't be represented within the size of the data type

Unsigned	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Normal Case

```
    010          2
+)  101      +)  5
-------      -----
    111          7
```

Overflow Case

```
    110          6
+)  101      +)  5
-------      -----
   1011         11
```

Unsigned Addition

- Similar to Decimal Addition
- Suppose we have a new data type that is 3-bit wide (c.f., `short` has 16 bits)
- Might overflow: result can't be represented within the size of the data type

Unsigned	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Normal Case

```
    010         2
+)  101    +)   5
-------    ------
    111         7
```

Overflow Case

```
    110         6
+)  101    +)   5
-------    ------
   1011        11   <---- True Sum
```

Unsigned Addition

- Similar to Decimal Addition
- Suppose we have a new data type that is 3-bit wide (c.f., **`short`** has 16 bits)
- Might overflow: result can't be represented within the size of the data type

Unsigned	Binary
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Normal Case

```
    010          2
+)  101      +)  5
-------      -----
    111          7
```

Overflow Case

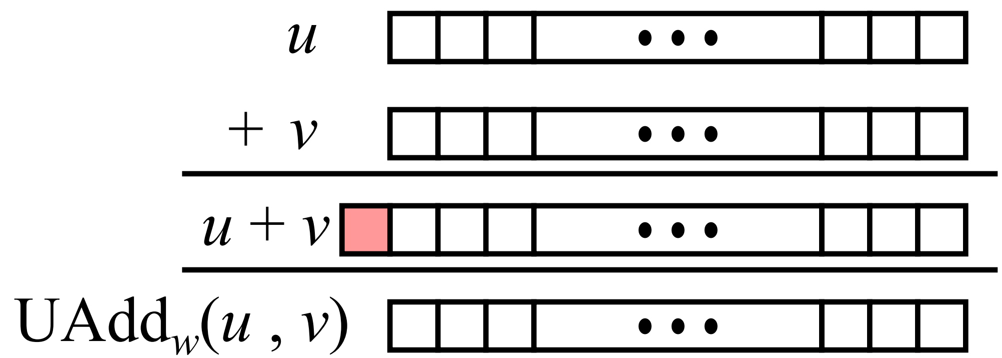
```
    110          6
+)  101      +)  5
-------      -----
   1011         11      True Sum
    011          3      Sum with same bits
```

Unsigned Addition in C

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



Two's Complement Addition

Signed	Binary
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

Two’s Complement Addition

- Has identical bit-level behavior as unsigned addition (a big advantage over sign-magnitude)

Signed	**Binary**
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

Two's Complement Addition

- Has identical bit-level behavior as unsigned addition (a big advantage over sign-magnitude)

Signed	Binary
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

Normal Case

```
    010          2
+)  101      +) -3
-------      -----
    111         -1
```

Two's Complement Addition

- Has identical bit-level behavior as unsigned addition (a big advantage over sign-magnitude)
- Overflow can also occur

Signed	Binary
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

Normal Case

```
    010         2
+)  101    +)  -3
-------    ------
    111        -1
```

Overflow Case

```
    110        -2
+)  101    +)  -3
-------    ------
   1011        -5
```

Two's Complement Addition

- Has identical bit-level behavior as unsigned addition (a big advantage over sign-magnitude)
- Overflow can also occur

Signed	Binary
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

Normal Case

```
    010          2
+)  101     +)  -3
-------     -------
    111         -1
```

Overflow Case

```
    110         -2
+)  101     +)  -3
-------     -------
   1011         -5
    011          3
```

Two's Complement Addition

- Has identical bit-level behavior as unsigned addition (a big advantage over sign-magnitude)
- Overflow can also occur

Signed	Binary
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

Min → -4

Normal Case

```
    010          2
+)  101     +)  -3
-------     -------
    111         -1
```

Overflow Case

```
    110         -2
+)  101     +)  -3
-------     -------
   1011         -5
    011          3
```

Negative Overflow

Two's Complement Addition

- Has identical bit-level behavior as unsigned addition (a big advantage over sign-magnitude)
- Overflow can also occur

Signed	Binary
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

Min → -4

Normal Case

```
     010          2
+)   101     +)  -3
    ----        ---
     111         -1
```

Overflow Case

```
     110         -2          011          3
+)   101     +)  -3     +)   001     +)   1
    ----        ---         ----         --
    1011         -5         0100          4
     011          3
```

Negative Overflow

Two's Complement Addition

- Has identical bit-level behavior as unsigned addition (a big advantage over sign-magnitude)
- Overflow can also occur

Signed	Binary
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

Min → -4 (100)

Normal Case

```
    010          2
+)  101      +) -3
-------      -----
    111         -1
```

Overflow Case

```
    110         -2          011          3
+)  101      +) -3      +)  001      +)  1
-------      -----      -------      -----
   1011         -5         0100          4
    011          3          100         -4
```

Negative Overflow

Two’s Complement Addition

- Has identical bit-level behavior as unsigned addition (a big advantage over sign-magnitude)
- Overflow can also occur

Signed	Binary
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

Max → 3

Min → -4

Normal Case

```
    010        2
+)  101    +) -3
-------    -----
    111       -1
```

Overflow Case

```
    110       -2        011        3
+)  101    +) -3    +)  001    +)  1
-------    -----    -------    -----
   1011       -5       0100        4
    011        3        100       -4
```

Negative Overflow

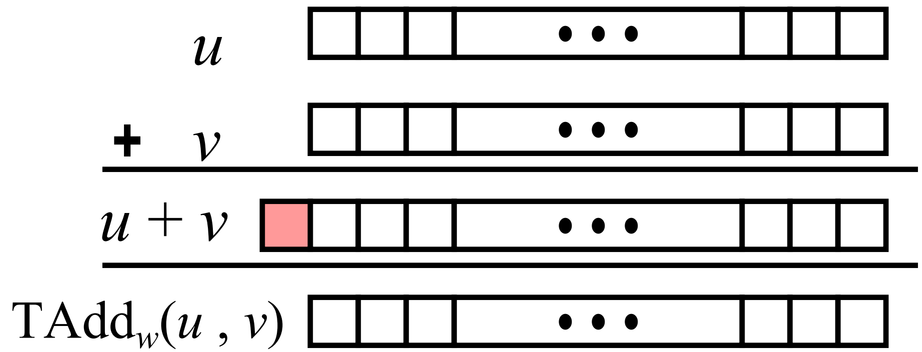
Positive Overflow

Two's Complement Addition in C

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



Is This Signed Addition an Overflow?

```
    111
+)  110
-------
   1101
```

Signed	Binary
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

Is This Signed Addition an Overflow?

```
    111
+)  110
   1101
```

Signed	Binary
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

Is This Signed Addition an Overflow?

```
   111
+) 110
------
  1101
```

Truncate

Signed	Binary
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

Is This Signed Addition an Overflow?

```
    111
+)  110
-------
  1101
```

Truncate →

```
     -1
+)   -2
-------
     -3
```

Signed	Binary
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

Is This Signed Addition an Overflow?

```
    111                          -1
+)  110                      +)  -2
-------        Truncate      ------
   1101       --------->         -3
```

Signed	Binary
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

- This is not an overflow by definition

Is This Signed Addition an Overflow?

```
    111
+)  110
-------
   1101
```

Truncate

```
     -1
+)   -2
-------
     -3
```

Signed	Binary
0	000
1	001
2	010
3	011
-4	100
-3	101
-2	110
-1	111

- This is not an overflow by definition
- Because the actual result can be represented using the bit width of the datatype (3 bits here)

Multiplication

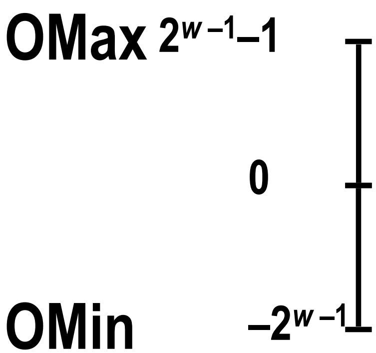
Multiplication

- Goal: Computing Product of w-bit numbers x, y

Multiplication

- Goal: Computing Product of *w*-bit numbers *x*, *y*

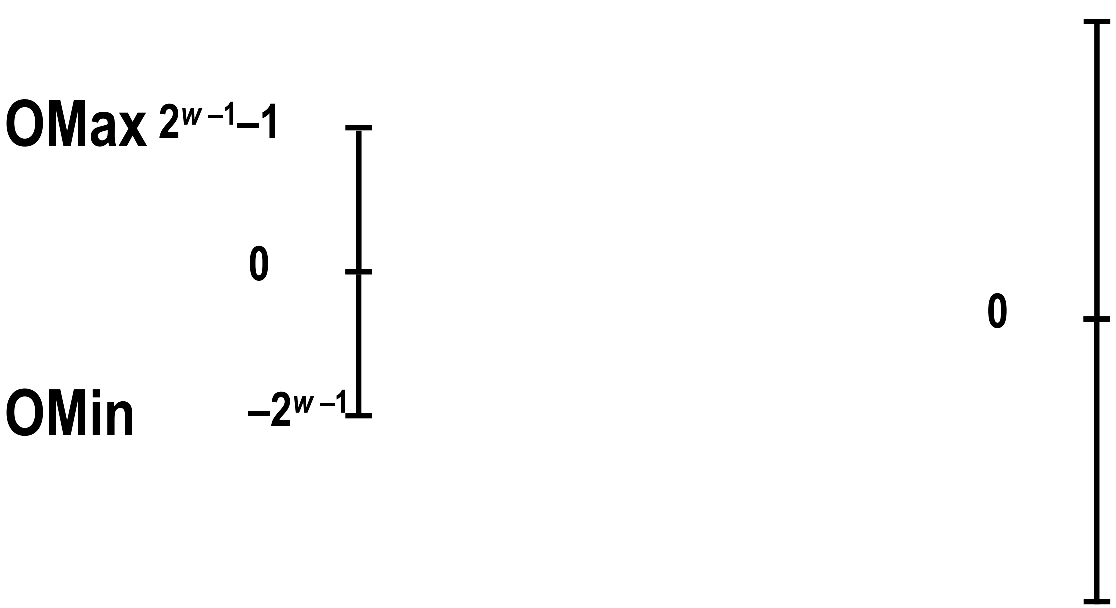
Original Number (w bits)



Multiplication

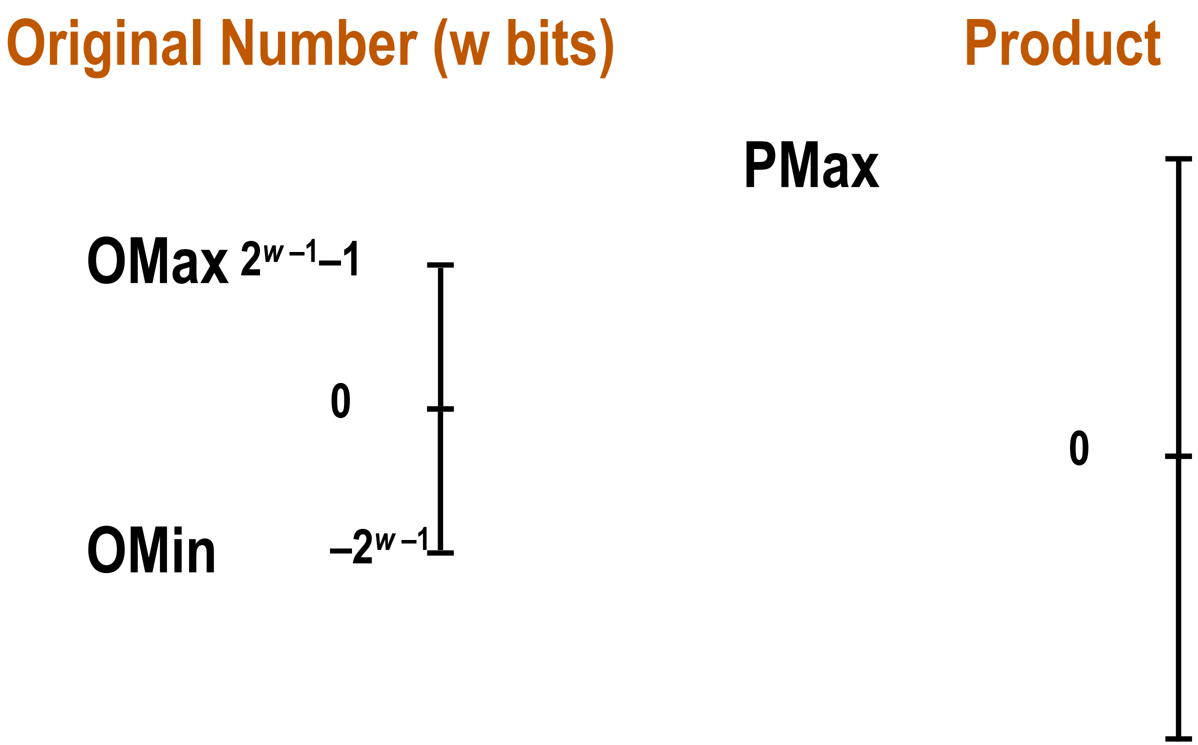
- Goal: Computing Product of *w*-bit numbers *x*, *y*

Original Number (w bits) **Product**



Multiplication

- Goal: Computing Product of *w*-bit numbers *x*, *y*



Multiplication

- Goal: Computing Product of w-bit numbers x, y

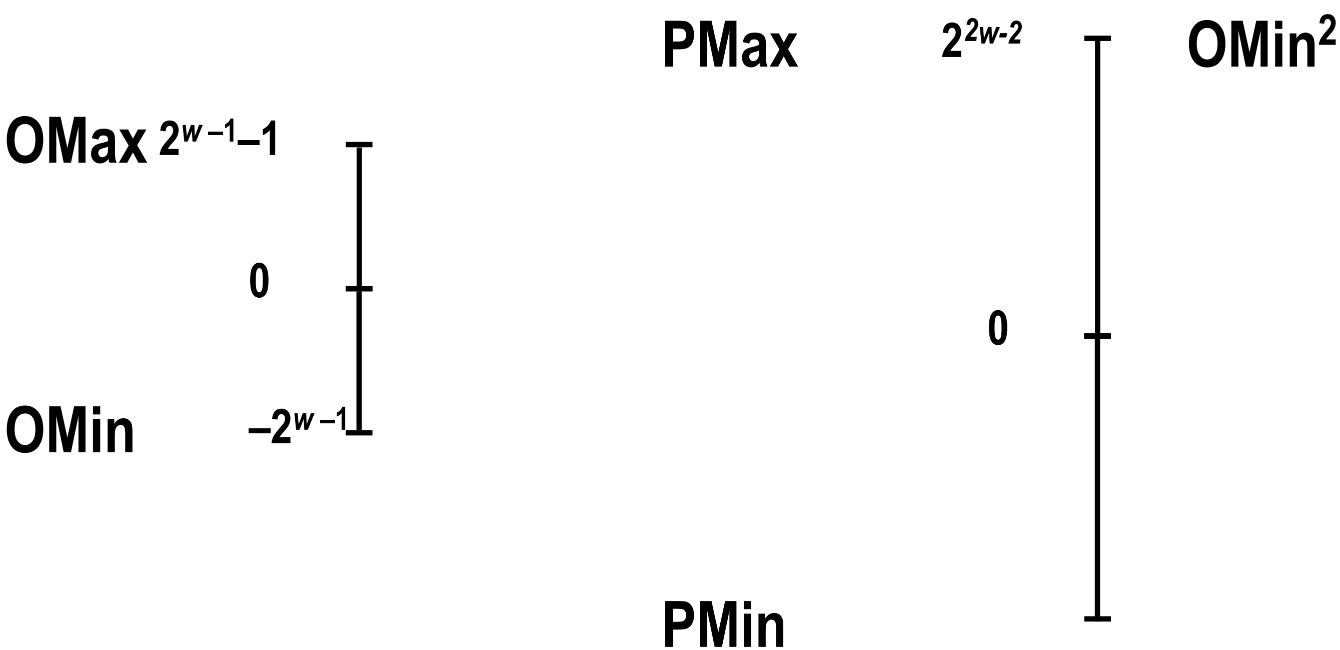
Original Number (w bits) **Product**

Multiplication

- Goal: Computing Product of *w*-bit numbers *x*, *y*

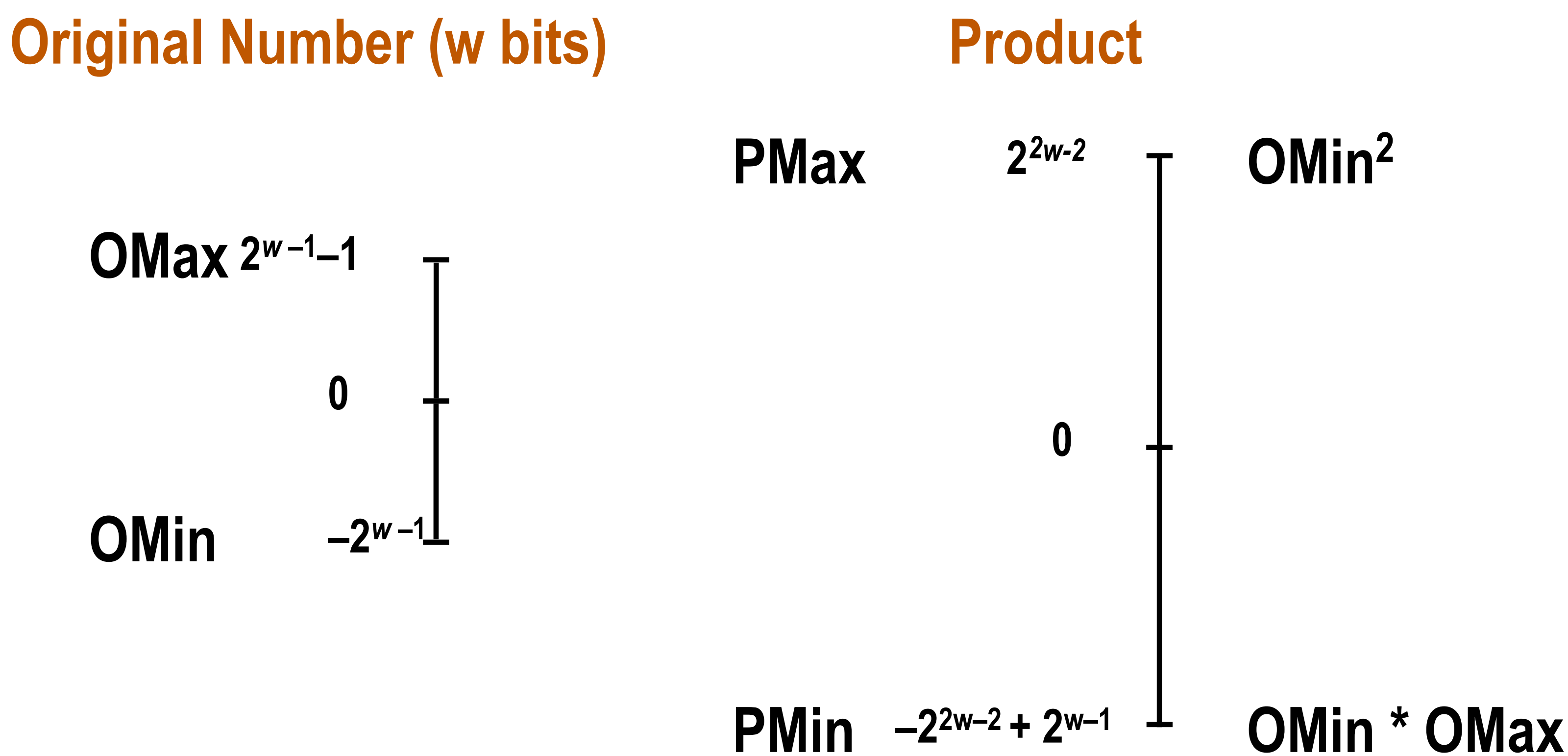
Original Number (w bits)

Product



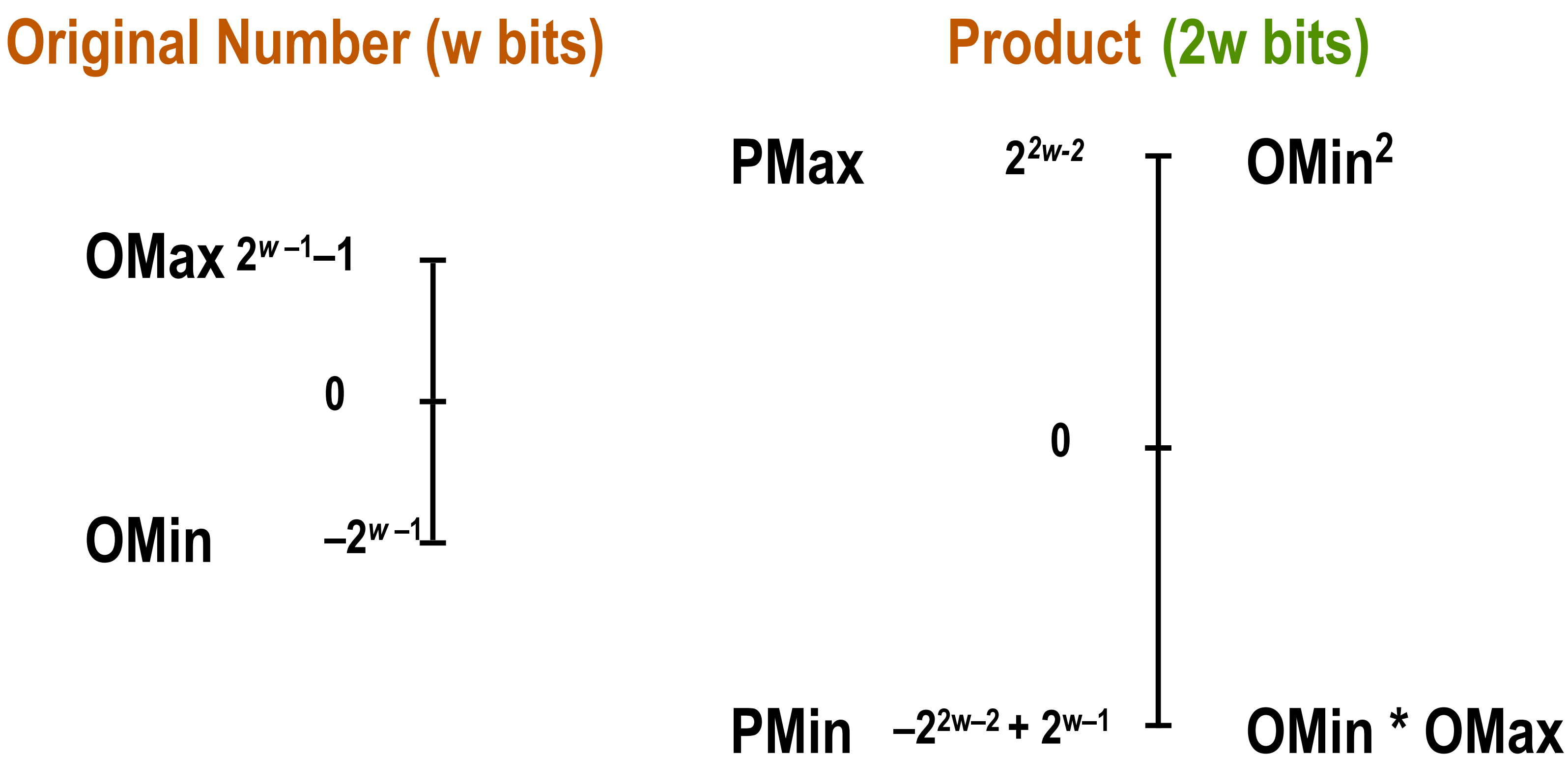
Multiplication

- Goal: Computing Product of w-bit numbers x, y



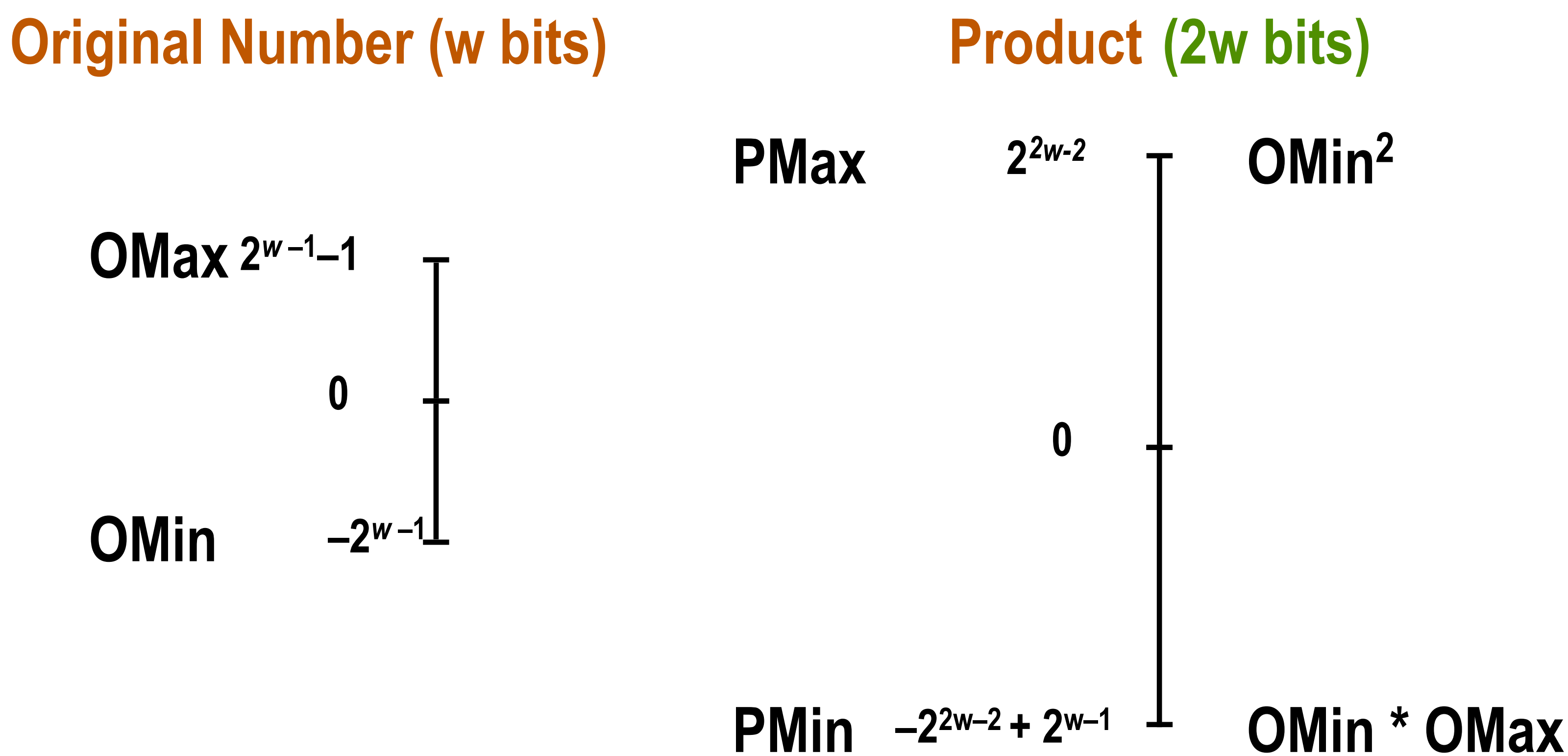
Multiplication

- Goal: Computing Product of *w*-bit numbers *x*, *y*



Multiplication

- Goal: Computing Product of *w*-bit numbers *x*, *y*
- Exact results can be bigger than *w* bits
  - Up to *2w* bits (both signed and unsigned)

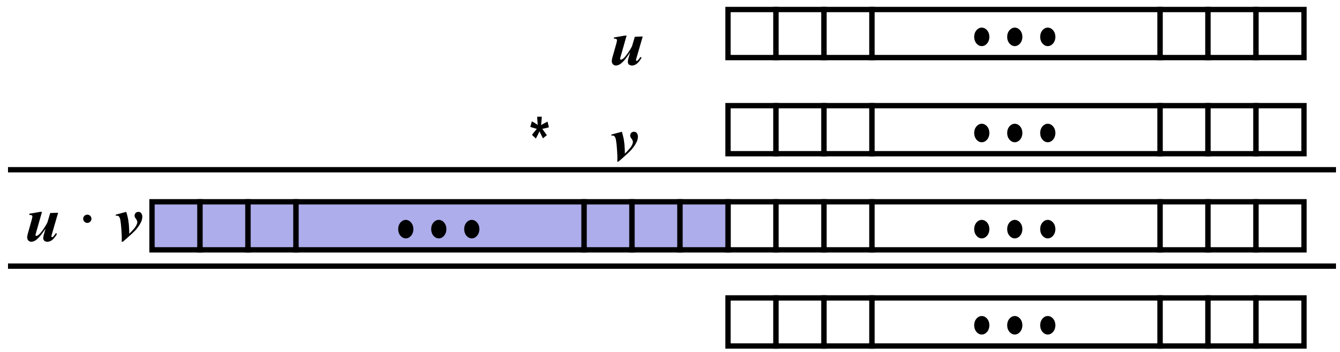


Unsigned Multiplication in C

Operands: *w* bits

True Product: 2*w bits

Discard *w* bits: *w* bits



- Standard Multiplication Function
  - Ignores high order w bits
- Effectively Implements the following:

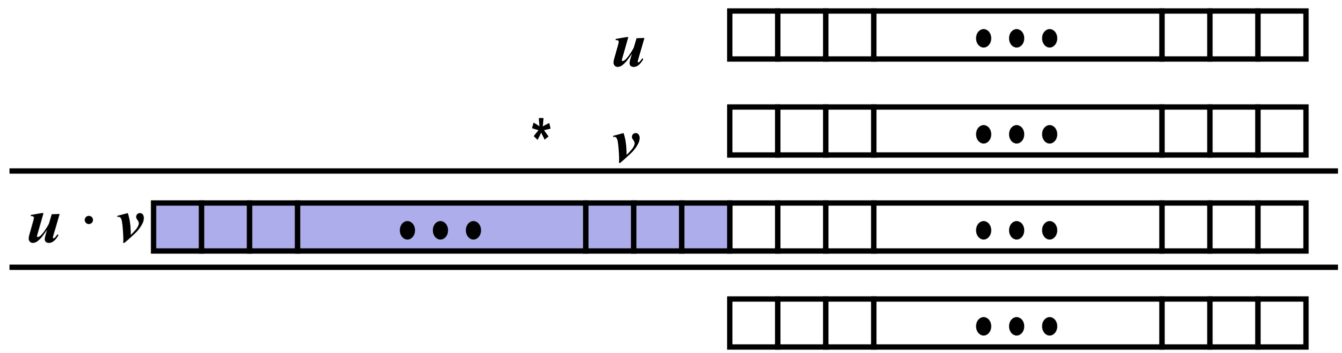
$\text{UMult}_w(u, v) = u \cdot v \bmod 2^w$

Unsigned Multiplication in C

Operands: *w* bits

True Product: 2w* bits**

Discard *w* bits: *w* bits

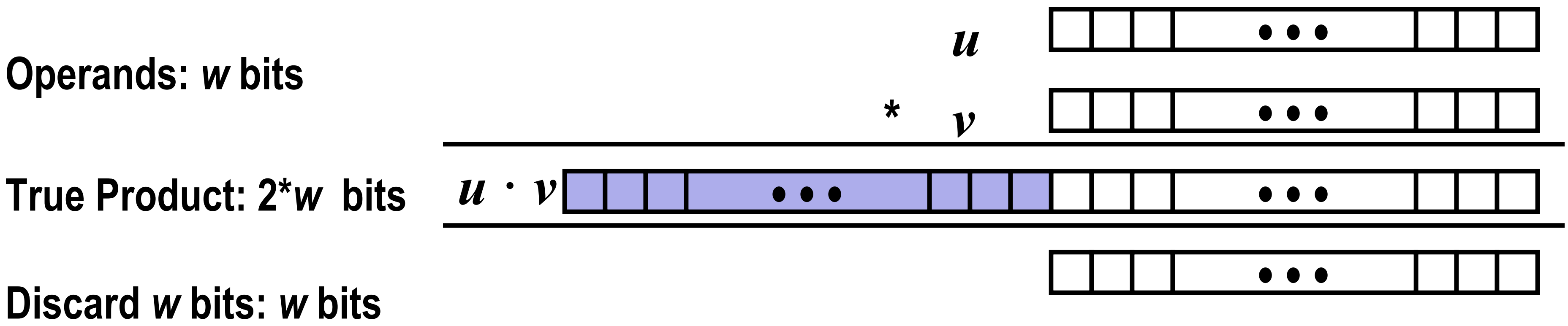


- Standard Multiplication Function
  - Ignores high order *w* bits
- Effectively Implements the following:

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

```
             1110 1001        E9          223
*            1101 0101     *  D5      *   213
 **** **** 1101 1101       C1DD       47499
             1101 1101        DD          221
```

Signed Multiplication in C



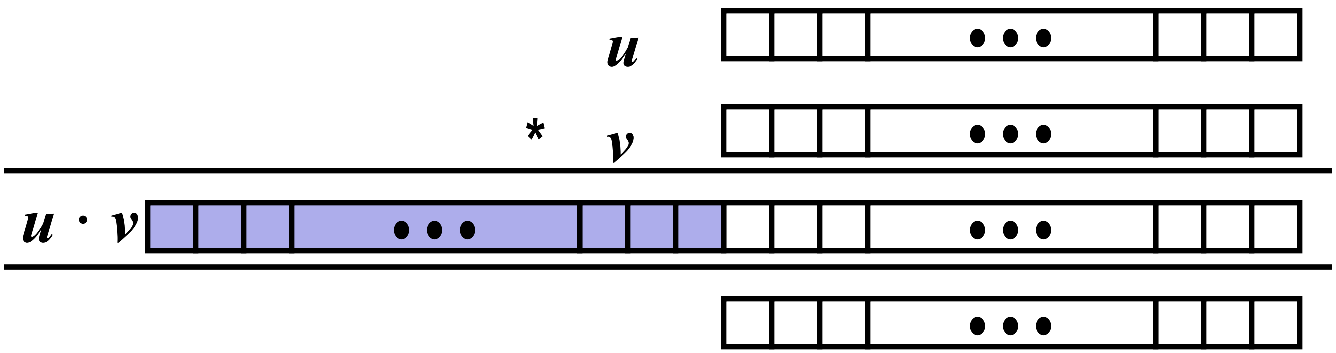
- Standard Multiplication Function
  - Ignores high order w bits
  - Some of which are different for signed vs. unsigned multiplication
  - Lower bits are the same

Signed Multiplication in C

Operands: *w* bits

True Product: 2w* bits**

Discard *w* bits: *w* bits



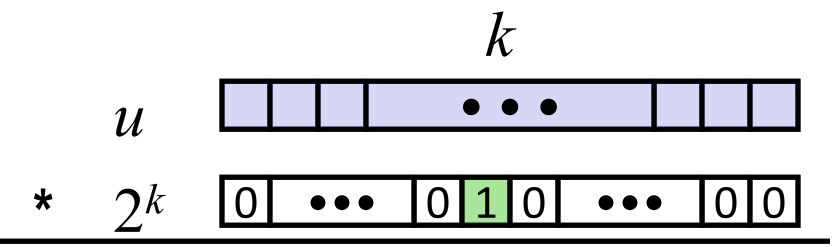
- Standard Multiplication Function
  - Ignores high order w bits
  - Some of which are different for signed vs. unsigned multiplication
  - Lower bits are the same

```
            1110 1001       E9        -23
*           1101 0101   *   D5   *    -43
 **** **** 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

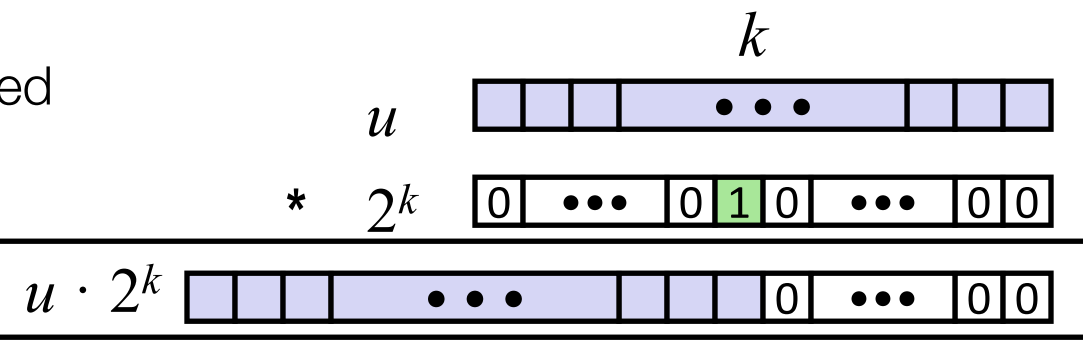


Power-of-2 Multiply with Shift

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

Operands: w bits

True Product: $w+k$ bits



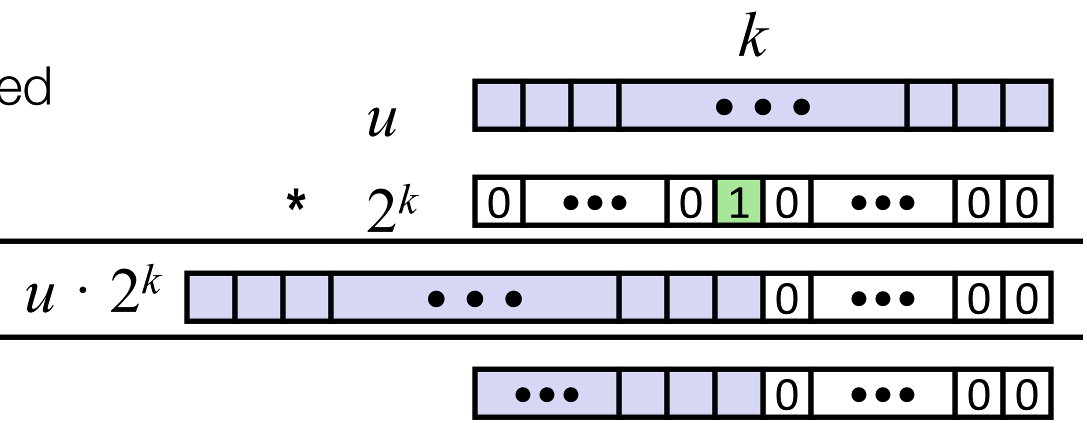
Power-of-2 Multiply with Shift

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

Operands: w bits

True Product: $w+k$ bits

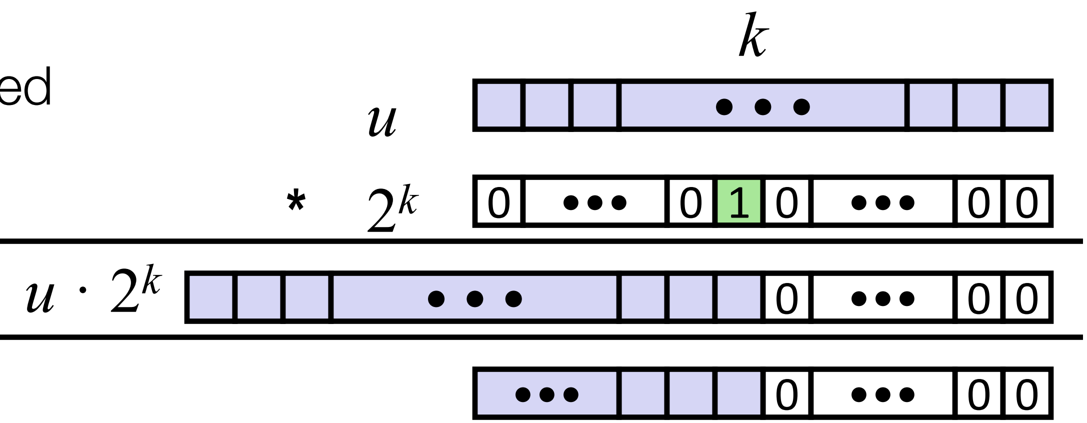
Discard k bits: w bits



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

Today: Representing Information in Binary

- Why Binary (bits)?
- Bit-level manipulations
- Integers
  - Representation: unsigned and signed
  - Conversion, casting
  - Expanding, truncating
  - Addition, negation, multiplication, shifting
  - Summary

Arithmetic: Basic Rules

- Addition:
  - Unsigned/signed: Normal addition followed by truncate, same operation on bit level

- Multiplication:
  - Unsigned/signed: Normal multiplication followed by truncate, same operation on bit level
  - Shift: Power-of-2 Multiply

Why Should I Use Unsigned?

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

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

  - Can be very subtle

```
#define DELTA sizeof(int)
int i;
for (i = CNT; i-DELTA >= 0; i-= DELTA)
  . . .
```

Why Should I Use Unsigned? – Bit Set

- *Use bits to represent my availability of the week*

b_6	b_5	b_4	b_3	b_2	b_1	b_0
Sun	Mon	Tue	Wed	Thu	Fri	Sat
1	0	1	1	0	0	1

- Use 1 bit per day, 7 bits in total.
- If bit x is set to 1 then I’m available on day mapped to bit x.

Why Should I Use Unsigned? – Bit Set

- *Use bits to represent my availability of the week*

b_6	b_5	b_4	b_3	b_2	b_1	b_0
Sun	**Mon**	**Tue**	**Wed**	**Thu**	**Fri**	**Sat**
1	**0**	**1**	**1**	**0**	**0**	**1**

- Use 1 bit per day, 7 bits in total.
- If bit x is set to 1 then I'm available on day mapped to bit x.

- In C: `unsigned int aval;`

Why Should I Use Unsigned? – Bit Set

- *Use bits to represent my availability of the week*

b_6	b_5	b_4	b_3	b_2	b_1	b_0
Sun	Mon	Tue	Wed	Thu	Fri	Sat
1	0	1	1	0	0	1

- Use 1 bit per day, 7 bits in total.
- If bit x is set to 1 then I'm available on day mapped to bit x.

- In C: `unsigned int aval;`

$$aval = 1*2^0 + 0*2^1 + 0*2^2 + 1*2^3 + 1*2^4 + 0*2^5 + 1*2^6 = 89_{10}$$

Today: Floating Point

- Background: Fractional binary numbers and fixed-point
- Floating point representation
- IEEE 754 standard
- Rounding, addition, multiplication
- Floating point in C
- Summary

Can We Represent Fractions in Binary?

- What does 10.01_2 mean?
  - C.f., Decimal

Can We Represent Fractions in Binary?

- What does 10.01_2 mean?
  - C.f., Decimal

$$12.45 = 1*10^{1} + 2*10^{0} + 4*10^{-1} + 5*10^{-2}$$

Can We Represent Fractions in Binary?

- What does 10.01_2 mean?
  - C.f., Decimal

$$12.45 = \boxed{1*10^1} + 2*10^0 + 4*10^{-1} + 5*10^{-2}$$

Can We Represent Fractions in Binary?

- What does 10.01_2 mean?
  - C.f., Decimal

$$12.45 = \boxed{1*10^1} + 2*10^0 + 4*10^{-1} + 5*10^{-2}$$

Can We Represent Fractions in Binary?

- What does 10.01_2 mean?
  - C.f., Decimal

$$12.45 = \boxed{1*10^1} + 2*10^0 + 4*10^{-1} + 5*10^{-2}$$

Can We Represent Fractions in Binary?

- What does 10.01_2 mean?
  - C.f., Decimal

$$12.45 = \boxed{1*10^1} + 2*10^0 + 4*10^{-1} + 5*10^{-2}$$

Can We Represent Fractions in Binary?

- What does 10.01_2 mean?
  - C.f., Decimal

$$12.45 = 1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2}$$

$$10.01_2 = 1*2^1 + 0*2^0 + 0*2^{-1} + 1*2^{-2}$$

Can We Represent Fractions in Binary?

- What does 10.01_2 mean?
  - C.f., Decimal

$$12.45 = 1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2}$$

$$10.01_2 = \boxed{1*2^1} + 0*2^0 + 0*2^{-1} + 1*2^{-2}$$

Can We Represent Fractions in Binary?

- What does 10.01_2 mean?
  - C.f., Decimal

$$12.45 = 1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2}$$

$$10.01_2 = \boxed{1*2^1} + 0*2^0 + 0*2^{-1} + 1*2^{-2}$$

Can We Represent Fractions in Binary?

- What does 10.01_2 mean?
  - C.f., Decimal

$$12.45 = 1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2}$$

$$10.01_2 = \boxed{1*2^1} + 0*2^0 + 0*2^{-1} + 1*2^{-2}$$

Can We Represent Fractions in Binary?

- What does 10.01_2 mean?
  - C.f., Decimal

$$12.45 = 1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2}$$

$$10.01_2 = \boxed{1*2^1} + 0*2^0 + 0*2^{-1} + 1*2^{-2}$$

Can We Represent Fractions in Binary?

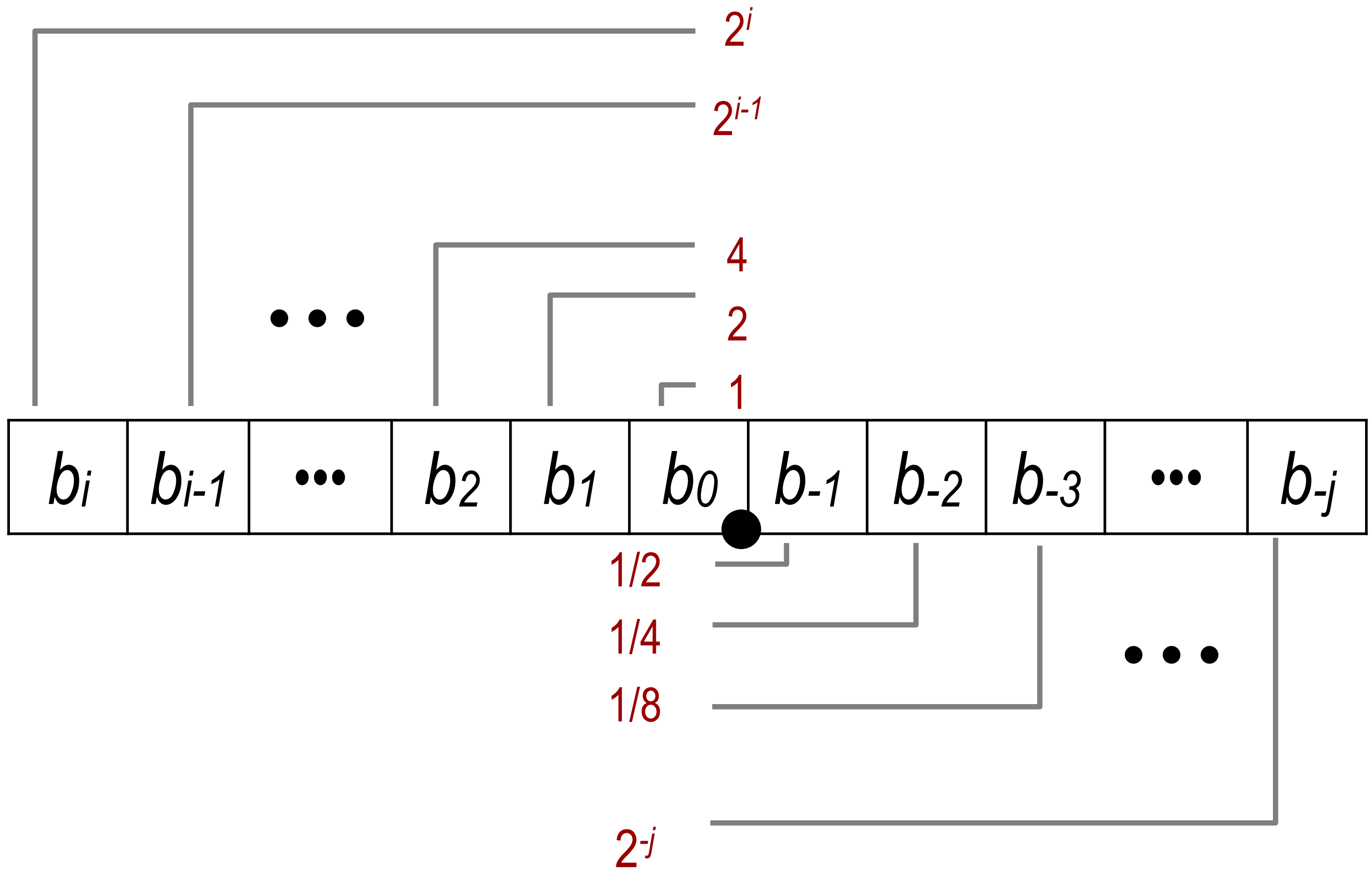
- What does 10.01_2 mean?
  - C.f., Decimal

$$12.45 = 1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2}$$

$$10.01_2 = 1*2^1 + 0*2^0 + 0*2^{-1} + 1*2^{-2}$$

$$= 2.25_{10}$$

Fractional Binary Numbers

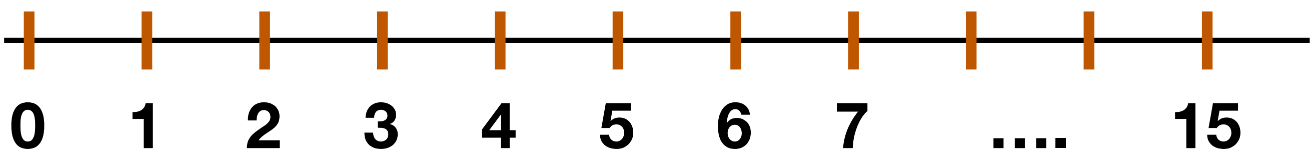


Can We Represent Fractions in Binary?

- What does 10.01_2 mean?
- C.f., Decimal
  - $12.45 = 1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2}$
- $10.01_2 = 1*2^1 + 0*2^0 + 0*2^{-1} + 1*2^{-2} = 2.25_{10}$

Can We Represent Fractions in Binary?

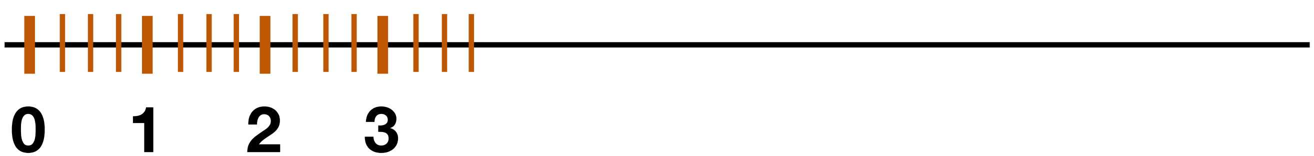
- What does 10.01_2 mean?
- C.f., Decimal
  - $12.45 = 1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2}$
- $10.01_2 = 1*2^1 + 0*2^0 + 0*2^{-1} + 1*2^{-2} = 2.25_{10}$



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

Can We Represent Fractions in Binary?

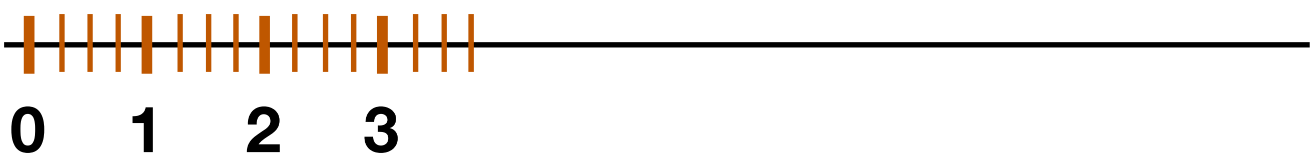
- What does 10.01_2 mean?
- C.f., Decimal
  - $12.45 = 1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2}$
- $10.01_2 = 1*2^1 + 0*2^0 + 0*2^{-1} + 1*2^{-2} = 2.25_{10}$



Decimal	Binary
0	00.00
0.25	00.01
0.5	00.10
0.75	00.11
1	01.00
1.25	01.01
1.5	01.10
1.75	01.11
2	10.00
2.25	10.01
2.5	10.10
2.75	10.11
3	11.00
3.25	11.01
3.5	11.10
3.75	11.11

Can We Represent Fractions in Binary?

- What does 10.01_2 mean?
- C.f., Decimal
  - $12.45 = 1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2}$
- $10.01_2 = 1*2^1 + 0*2^0 + 0*2^{-1} + 1*2^{-2} = 2.25_{10}$



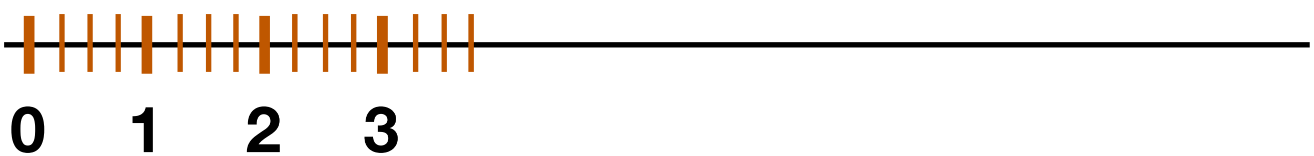

```
   01.10
+  01.01
--------
   10.11
```

```
   1.50
+  1.25
-------
   2.75
```

Decimal	Binary
0	00.00
0.25	00.01
0.5	00.10
0.75	00.11
1	01.00
1.25	01.01
1.5	01.10
1.75	01.11
2	10.00
2.25	10.01
2.5	10.10
2.75	10.11
3	11.00
3.25	11.01
3.5	11.10
3.75	11.11

Can We Represent Fractions in Binary?

- What does 10.01_2 mean?
- C.f., Decimal
  - $12.45 = 1*10^1 + 2*10^0 + 4*10^{-1} + 5*10^{-2}$
- $10.01_2 = 1*2^1 + 0*2^0 + 0*2^{-1} + 1*2^{-2} = 2.25_{10}$



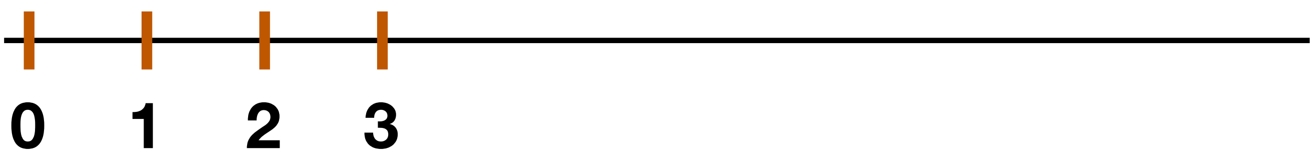
Integer Arithmetic Still Works!

```
  01.10          1.50
+ 01.01        + 1.25
-------        ------
  10.11          2.75
```

Decimal	Binary
0	00.00
0.25	00.01
0.5	00.10
0.75	00.11
1	01.00
1.25	01.01
1.5	01.10
1.75	01.11
2	10.00
2.25	10.01
2.5	10.10
2.75	10.11
3	11.00
3.25	11.01
3.5	11.10
3.75	11.11

Fixed-Point Representation

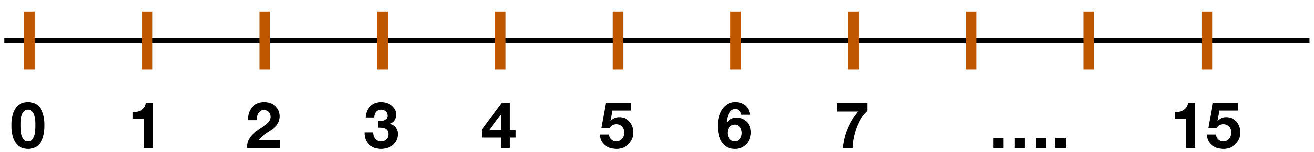
- Binary point stays fixed
- Fixed interval between two representable numbers as long as the binary point stays fixed
  - The interval in this example is 0.25_{10}
- Fixed-point representation of numbers
  - Integer is one special case of fixed-point



Decimal	Binary
0	00.00
0.25	00.01
0.5	00.10
0.75	00.11
1	01.00
1.25	01.01
1.5	01.10
1.75	01.11
2	10.00
2.25	10.01
2.5	10.10
2.75	10.11
3	11.00
3.25	11.01
3.5	11.10
3.75	11.11

Fixed-Point Representation

- Binary point stays fixed
- Fixed interval between two representable numbers as long as the binary point stays fixed
  - The interval in this example is 0.25_{10}
- Fixed-point representation of numbers
  - Integer is one special case of fixed-point



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

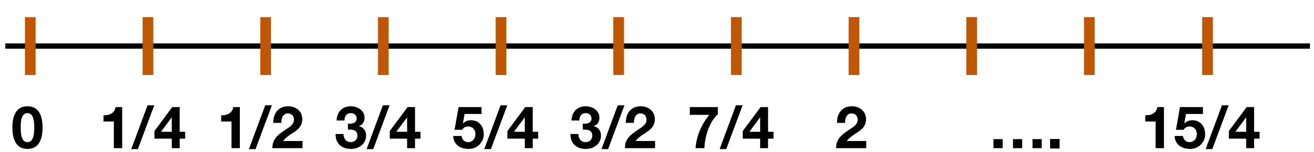
Limitations of Fixed-Point (#*1*)

Limitations of Fixed-Point (#*1*)

- Can exactly represent numbers only of the form $x/2^k$

Limitations of Fixed-Point (#*1*)

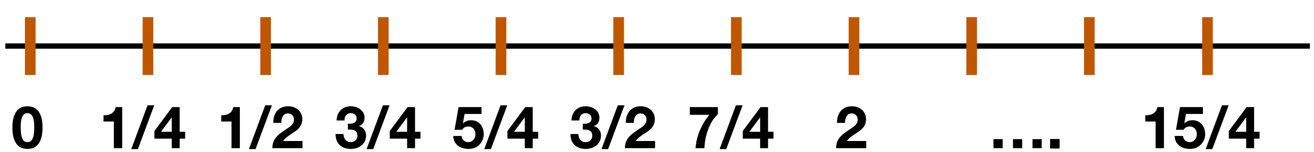
- Can exactly represent numbers only of the form $x/2^k$



$b_3b_2.b_1b_0$

Limitations of Fixed-Point (#*1*)

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

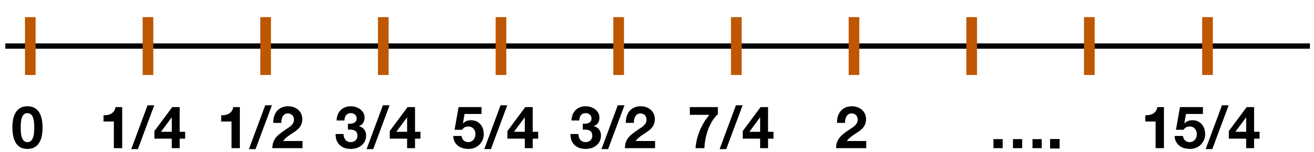


$b_3b_2.b_1b_0$

Limitations of Fixed-Point (#*1*)

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

Decimal Value	**Binary Representation**
1/3	0.0101010101[01]…
1/5	0.001100110011[0011]…
1/10	0.0001100110011[0011]…



$b_3b_2.b_1b_0$

Limitations of Fixed-Point (#2)

Limitations of Fixed-Point (#2)

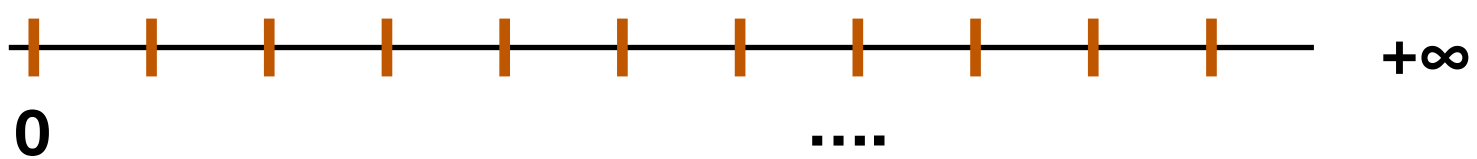
- Can’t represent very small and very large numbers at the same time

Limitations of Fixed-Point (#2)

- Can't represent very small and very large numbers at the same time
  - To represent very large numbers, the (fixed) interval needs to be large, making it hard to represent small numbers

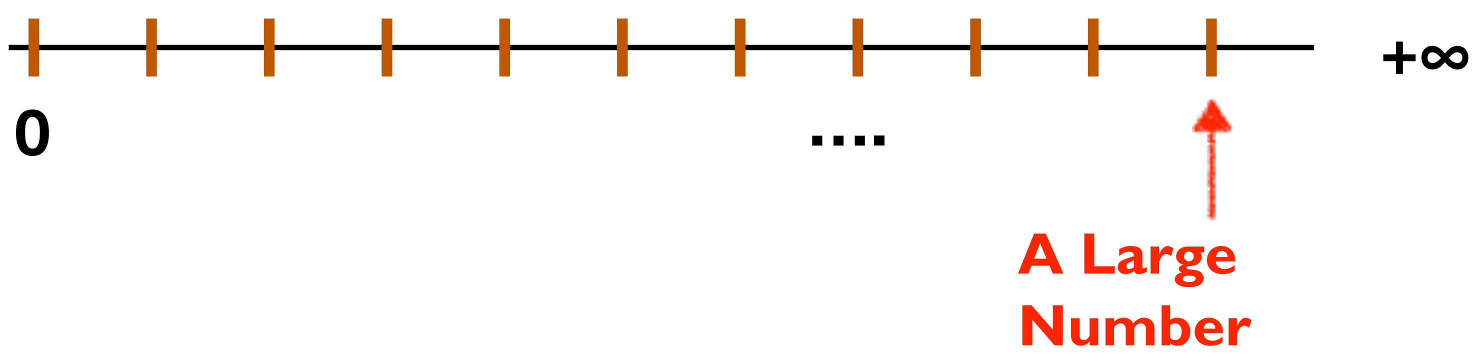
Limitations of Fixed-Point (#2)

- Can’t represent very small and very large numbers at the same time
  - To represent very large numbers, the (fixed) interval needs to be large, making it hard to represent small numbers



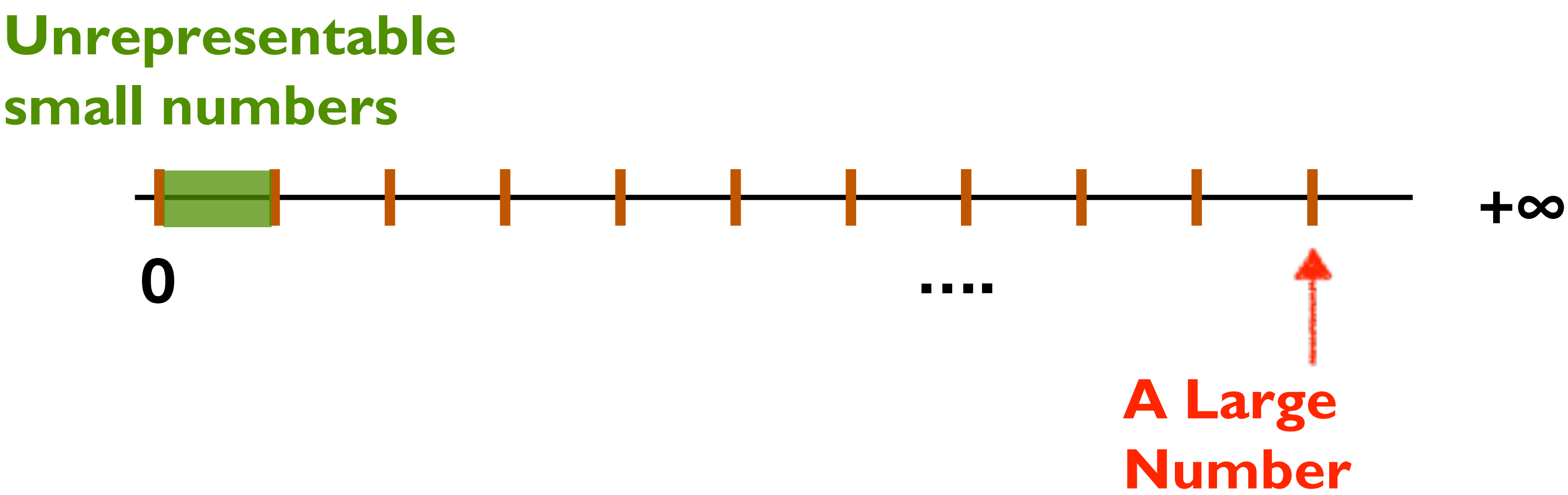
Limitations of Fixed-Point (#2)

- Can’t represent very small and very large numbers at the same time
  - To represent very large numbers, the (fixed) interval needs to be large, making it hard to represent small numbers



Limitations of Fixed-Point (#2)

- Can't represent very small and very large numbers at the same time
  - To represent very large numbers, the (fixed) interval needs to be large, making it hard to represent small numbers



Limitations of Fixed-Point (#2)

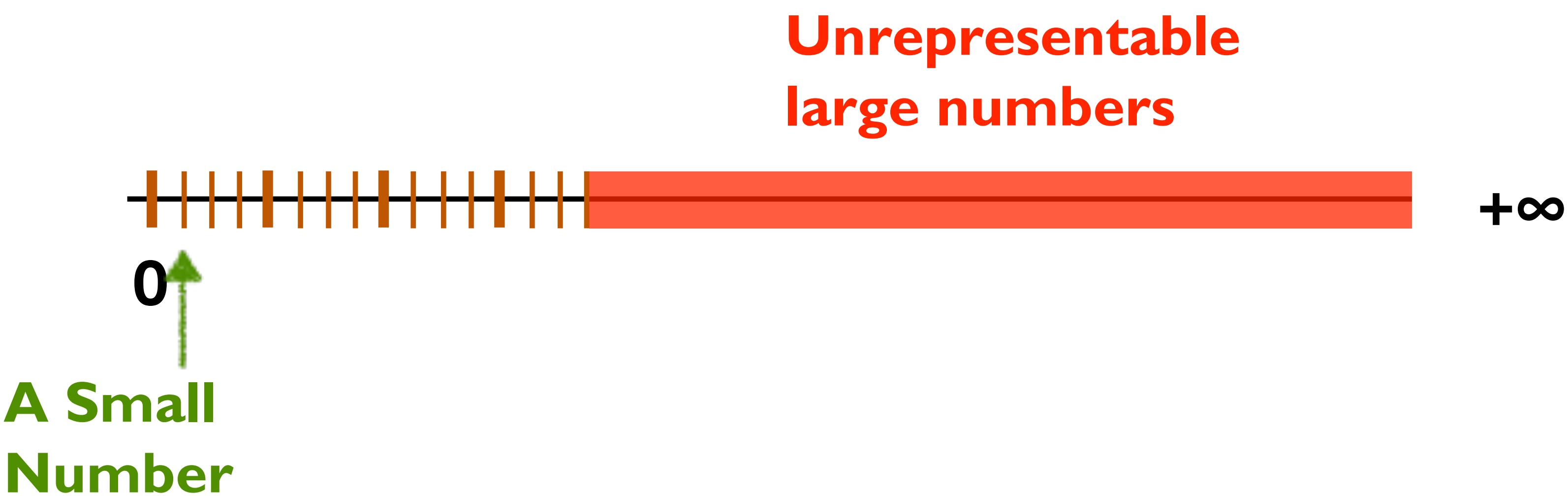
- Can't represent very small and very large numbers at the same time
  - To represent very large numbers, the (fixed) interval needs to be large, making it hard to represent small numbers
  - To represent very small numbers, the (fixed) interval needs to be small, making it hard to represent large numbers

Limitations of Fixed-Point (#2)

- Can't represent very small and very large numbers at the same time
  - To represent very large numbers, the (fixed) interval needs to be large, making it hard to represent small numbers
  - To represent very small numbers, the (fixed) interval needs to be small, making it hard to represent large numbers

Limitations of Fixed-Point (#2)

- Can’t represent very small and very large numbers at the same time
  - To represent very large numbers, the (fixed) interval needs to be large, making it hard to represent small numbers
  - To represent very small numbers, the (fixed) interval needs to be small, making it hard to represent large numbers



Today: Floating Point

- Background: Fractional binary numbers and fixed-point
- Floating point representation
- IEEE 754 standard
- Rounding, addition, multiplication
- Floating point in C
- Summary

Primer: (Normalized) Scientific Notation

- In decimal: $M \times 10^E$
  - *E* is an integer
  - Normalized form: $1 <= |M| < 10$

Primer: (Normalized) Scientific Notation

- In decimal: $M \times 10^E$
  - E is an integer
  - Normalized form: $1 <= |M| < 10$

Decimal Value	Scientific Notation
2	2×10^0
-4,321.768	-4.321768×10^3
0.000 000 007 51	7.51×10^{-9}

Primer: (Normalized) Scientific Notation

- In decimal: $M \times 10^E$
  - *E* is an integer
  - Normalized form: $1 <= |M| < 10$

$$M \times 10^E$$

Decimal Value	Scientific Notation
2	2×10^0
-4,321.768	-4.321768×10^3
0.000 000 007 51	7.51×10^{-9}

Primer: (Normalized) Scientific Notation

- In decimal: $M \times 10^E$
  - E is an integer
  - Normalized form: $1 <= |M| < 10$

$$M \times 10^E$$

Significand

Decimal Value	Scientific Notation
2	2×10^0
-4,321.768	-4.321768×10^3
0.000 000 007 51	7.51×10^{-9}

Primer: (Normalized) Scientific Notation

- In decimal: $M \times 10^E$
  - E is an integer
  - Normalized form: $1 <= |M| < 10$

$$M \times 10^E$$

Significand Base

Decimal Value	Scientific Notation
2	2×10^0
-4,321.768	-4.321768×10^3
0.000 000 007 51	7.51×10^{-9}

Primer: (Normalized) Scientific Notation

- In decimal: $M \times 10^E$
  - E is an integer
  - Normalized form: $1 <= |M| < 10$

$$\underbrace{M}_{\text{Significand}} \times \underbrace{10}_{\text{Base}}{}^{\overset{\text{Exponent}}{E}}$$

Decimal Value	Scientific Notation
2	2×10^0
-4,321.768	-4.321768×10^3
0.000 000 007 51	7.51×10^{-9}

Primer: (Normalized) Scientific Notation

- In binary: $(-1)^s\, M\, 2^E$
- Normalized form:
  - $1 <= M < 2$
  - $M = 1.\underbrace{b_0 b_1 b_2 b_3 \ldots}_{\text{Fraction}}$

$$(-1)^{\overset{\text{Sign}}{s}}\ \underset{\text{Significand}}{M} \times \underset{\text{Base}}{2}^{\overset{\text{Exponent}}{E}}$$

Binary Value	Scientific Notation
1110110110110	$(-1)^0\ 1.110110110110 \times 2^{12}$
-101.11	$(-1)^1\ 1.0111 \times 2^2$
0.00101	$(-1)^0\ 1.01 \times 2^{-3}$

Primer: (Normalized) Scientific Notation

- In binary: $(-1)^s M\ 2^E$
- Normalized form:
  - $1 <= M < 2$
  - $M = 1.\underbrace{b_0b_1b_2b_3\ldots}_{\text{Fraction}}$

$$(-1)^{\overset{\text{Sign}}{s}}\ \underset{\text{Significand}}{M} \times \underset{\text{Base}}{2}^{\overset{\text{Exponent}}{E}}$$

- If I tell you that there is a number where:
  - Fraction = 0101
  - s = 1
  - E = 10
  - You could reconstruct the number as $(-1)^1 1.0101\text{x}2^{10}$

Primer: Floating Point Representation

- In binary: $(-1)^s\ M\ 2^E$
- Normalized form:
  - $1 <= M < 2$
  - $M = 1.b_0b_1b_2b_3\ldots$

Fraction

Sign

Exponent

$$(-1)^{s}\ M \times 2^{E}$$

Significand

Base

Primer: Floating Point Representation

- In binary: $(-1)^s\ M\ 2^E$
- Normalized form:
  - $1 <= M < 2$
  - $M = 1.\underbrace{b_0 b_1 b_2 b_3 \ldots}_{\text{Fraction}}$
- **Encoding**

Sign Exponent

$$(-1)^{s}\ M \times 2^{E}$$

Significand Base

Primer: Floating Point Representation

- In binary: $(-1)^s\ M\ 2^E$
- Normalized form:
  - $1 <= M < 2$
  - $M = 1.\underbrace{b_0 b_1 b_2 b_3 \ldots}_{\text{Fraction}}$
- **Encoding**

$$(-1)^{\overset{\text{Sign}}{s}}\ \underset{\text{Significand}}{M} \times \underset{\text{Base}}{2}^{\overset{\text{Exponent}}{E}}$$

Primer: Floating Point Representation

- In binary: $(-1)^s\, M\, 2^E$
- Normalized form:
  - $1 <= M < 2$
  - $M = 1.b_0b_1b_2b_3\ldots$

    Fraction
- **Encoding**
  - MSB ***s*** is sign bit s

Sign Exponent

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

Significand Base

Primer: Floating Point Representation

- In binary: $(-1)^s \, M \, 2^E$
- Normalized form:
  - $1 <= M < 2$
  - $M = 1.b_0b_1b_2b_3\ldots$

    Fraction
- **Encoding**
  - MSB ***s*** is sign bit s
  - ***exp*** field encodes Exponent (but not exactly the same, more later)

Sign Exponent

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

Significand Base

Primer: Floating Point Representation

- In binary: $(-1)^s\, M\, 2^E$
- Normalized form:
  - $1 <= M < 2$
  - $M = 1.b_0b_1b_2b_3\ldots$

    Fraction

Sign Exponent

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

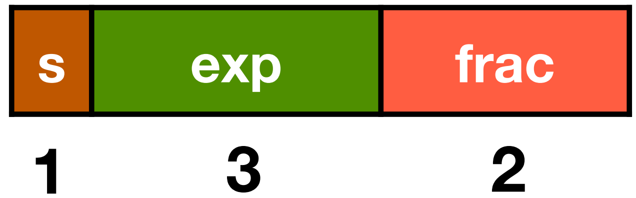
Significand Base

- **Encoding**
  - MSB ***s*** is sign bit s
  - ***exp*** field encodes Exponent (but not exactly the same, more later)
  - ***frac*** field encodes Fraction (but not exactly the same, more later)

s	exp	frac

6-bit Floating Point Example

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



6-bit Floating Point Example

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

- *exp* has 3 bits, interpreted as an unsigned value

6-bit Floating Point Example

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

s	exp	frac
1	3	2

- *exp* has 3 bits, interpreted as an unsigned value
  - If ***exp*** were ***E***, we could represent exponents from **0 to 7**

6-bit Floating Point Example

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

s	exp	frac
1	**3**	**2**

- *exp* has 3 bits, interpreted as an unsigned value
  - If ***exp*** were ***E***, we could represent exponents from **0 to 7**
  - How about negative exponent?

6-bit Floating Point Example

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

s	exp	frac
1	3	2

- *exp* has 3 bits, interpreted as an unsigned value
  - If ***exp*** were ***E***, we could represent exponents from **0 to 7**
  - How about negative exponent?
  - Subtract a bias term: ***E = exp - bias*** (i.e., exp = E + bias)

6-bit Floating Point Example

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

s	exp	frac
1	3	2

- *exp* has 3 bits, interpreted as an unsigned value
  - If ***exp*** were ***E***, we could represent exponents from **0 to 7**
  - How about negative exponent?
  - Subtract a bias term: ***E = exp - bias*** (i.e., exp = E + bias)
  - bias is always $2^{k-1} - 1$, where k is number of exponent bits

6-bit Floating Point Example

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

s	exp	frac
1	3	2

- *exp* has 3 bits, interpreted as an unsigned value
  - If ***exp*** were ***E***, we could represent exponents from **0 to 7**
  - How about negative exponent?
  - Subtract a bias term: ***E = exp - bias*** (i.e., exp = E + bias)
  - bias is always $2^{k-1} - 1$, where k is number of exponent bits
- Example when we use 3 bits for exp (i.e., $k = 3$):

6-bit Floating Point Example

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

s	exp	frac
1	3	2

- *exp* has 3 bits, interpreted as an unsigned value
  - If ***exp*** were ***E***, we could represent exponents from **0 to 7**
  - How about negative exponent?
  - Subtract a bias term: ***E = exp - bias*** (i.e., exp = E + bias)
  - bias is always $2^{k-1} - 1$, where k is number of exponent bits
- Example when we use 3 bits for exp (i.e., $k = 3$):
  - bias = 3

6-bit Floating Point Example

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

s	exp	frac
1	3	2

- *exp* has 3 bits, interpreted as an unsigned value
  - If ***exp*** were ***E***, we could represent exponents from **0 to 7**
  - How about negative exponent?
  - Subtract a bias term: ***E = exp - bias*** (i.e., exp = E + bias)
  - bias is always $2^{k-1} - 1$, where k is number of exponent bits
- Example when we use 3 bits for exp (i.e., $k = 3$):
  - bias = 3
  - If ***E*** = -2, ***exp*** is 1 (001_2)

6-bit Floating Point Example

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

s	exp	frac
1	3	2

- *exp* has 3 bits, interpreted as an unsigned value
  - If ***exp*** were ***E***, we could represent exponents from **0 to 7**
  - How about negative exponent?
  - Subtract a bias term: ***E = exp - bias*** (i.e., exp = E + bias)
  - bias is always $2^{k-1} - 1$, where k is number of exponent bits
- Example when we use 3 bits for exp (i.e., $k = 3$):
  - bias = 3
  - If ***E*** = -2, ***exp*** is 1 (001_2)

E	exp
-3	000
-2	001
-1	010
0	011
1	100
2	101
3	110
4	111

6-bit Floating Point Example

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

s	exp	frac
1	3	2

- *exp* has 3 bits, interpreted as an unsigned value
  - If ***exp*** were ***E***, we could represent exponents from **0 to 7**
  - How about negative exponent?
  - Subtract a bias term: ***E = exp - bias*** (i.e., exp = E + bias)
  - bias is always $2^{k-1} - 1$, where k is number of exponent bits
- Example when we use 3 bits for exp (i.e., $k = 3$):
  - bias = 3
  - If ***E*** = -2, ***exp*** is 1 (001_2)
  - Reserve 000 and 111 for other purposes (more on this later)

E	exp
~~-3~~	~~000~~
-2	001
-1	010
0	011
1	100
2	101
3	110
~~4~~	~~111~~

6-bit Floating Point Example

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

s	exp	frac
1	3	2

- *exp* has 3 bits, interpreted as an unsigned value
  - If ***exp*** were ***E***, we could represent exponents from **0 to 7**
  - How about negative exponent?
  - Subtract a bias term: ***E = exp - bias*** (i.e., exp = E + bias)
  - bias is always $2^{k-1} - 1$, where k is number of exponent bits
- Example when we use 3 bits for exp (i.e., *k* = 3):
  - bias = 3
  - If ***E*** = -2, ***exp*** is 1 (001_2)
  - Reserve 000 and 111 for other purposes (more on this later)
  - We can now represent exponents from **-2 (exp 001) to 3 (exp 110)**

E	exp
~~-3~~	~~000~~
-2	001
-1	010
0	011
1	100
2	101
3	110
~~4~~	~~111~~

6-bit Floating Point Example

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

s	exp	frac
1	3	2

- *frac* has 2 bits, append them after "1." to form M
  - ***frac*** = 10 implies M = 1.10

6-bit Floating Point Example

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

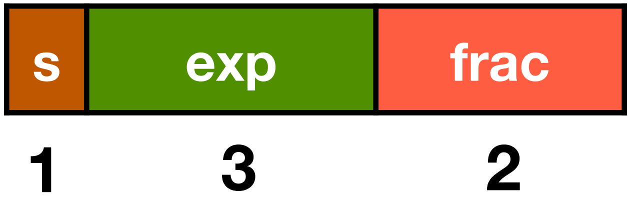
s	exp	frac
1	3	2

- *frac* has 2 bits, append them after “1.” to form M
    - ***frac*** = 10 implies M = 1.10
- Putting it Together: An Example:

$$-10.1_2 = (-1)^1\ 1.01 \times 2^1$$

6-bit Floating Point Example

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



- *frac* has 2 bits, append them after “1.” to form M
  - ***frac*** = 10 implies M = 1.10
- Putting it Together: An Example:

$$-10.1_2 = (-1)^1\, 1.01 \times 2^1$$

6-bit Floating Point Example

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

1	exp	frac
1	3	2

- *frac* has 2 bits, append them after “1.” to form M
  - ***frac*** = 10 implies M = 1.10
- Putting it Together: An Example:

$$-10.1_2 = (-1)^1 \, 1.01 \times 2^1$$

6-bit Floating Point Example

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

1	exp	frac
1	3	2

- *frac* has 2 bits, append them after “1.” to form M
  - ***frac*** = 10 implies M = 1.10
- Putting it Together: An Example:

$$-10.1_2 = (-1)^1\ 1.01 \times 2^1$$

6-bit Floating Point Example

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

1	exp	frac
1	3	2

- *frac* has 2 bits, append them after "1." to form M
  - ***frac*** = 10 implies M = 1.10
- Putting it Together: An Example:

$$-10.1_2 = (-1)^1 \, 1.01 \times 2^1$$

E	exp
~~-3~~	~~000~~
-2	001
-1	010
0	011
1	100
2	101
3	110
~~4~~	~~111~~

6-bit Floating Point Example

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

1	100	frac
1	3	2

- *frac* has 2 bits, append them after "1." to form M
  - ***frac*** = 10 implies M = 1.10
- Putting it Together: An Example:

$$-10.1_2 = (-1)^1\, 1.01 \times 2^1$$

E	exp
~~-3~~	~~000~~
-2	001
-1	010
0	011
1	100
2	101
3	110
~~4~~	~~111~~

6-bit Floating Point Example

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

1	**100**	**frac**
1	**3**	**2**

- *frac* has 2 bits, append them after "1." to form M
  - ***frac*** = 10 implies M = 1.10
- Putting it Together: An Example:

$$-10.1_2 = (-1)^1 \, 1.01 \times 2^1$$

E	exp
~~-3~~	~~000~~
-2	001
-1	010
0	011
1	100
2	101
3	110
~~4~~	~~111~~

6-bit Floating Point Example

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

1	100	01
1	3	2

- *frac* has 2 bits, append them after "1." to form M
  - ***frac*** = 10 implies M = 1.10
- Putting it Together: An Example:

$$-10.1_2 = (-1)^1 \, 1.01 \times 2^1$$

E	exp
~~-3~~	~~000~~
-2	001
-1	010
0	011
1	100
2	101
3	110
~~4~~	~~111~~