System Program :

data representation - int

Honguk Woo

Encoding Integers

- w-bit vector : $[x_{w-1}, x_{w-2}, ..., x_0]$

Unsigned (0, positive)

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

Binary to Unsigned

Two's Complement (negative, 0, positive)

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

Binary to Two's complement

Sign Bit
(most significant bit)

- e.g., 4bit binary -> integer
  - 1111
  - Unsigned $2^3+2^2+2^1+2^0 = 15$
  - Two's Complement

$-8+4+2+1 = -1$

$-(0001)$

Encoding Integers

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

```
short int x =  15213;   /* 2 byte long */
short int y = -15213;
```

Unsigned

x

Two's Complement

$y = 2^n - x$

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

Two’s Complement Encoding

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

Two’s Complement

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

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

- Unsigned, 4 bits

1111 (15, max)
:
:
0111 (7)
:
:
0000 (0, min)

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

- Two’s complement, 4 bits

0111 (7, max)
:
:
0000 (0)
1111 (-1)
:
:
1000 (-8, min)

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

Numeric Ranges

- Unsigned Values
  - *UMin* = 0
    000…0
  - *UMax* = $2^w - 1$
    111…1
    2^{w-1}
- Two's Complement Values
  - *TMin* = -2^{w-1}
    100…0
  - *TMax* = $2^{w-1} - 1$
    011…1
- Other Values
  - -1
    111…1

Values for *W* = 16

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

Values for Different Word Sizes

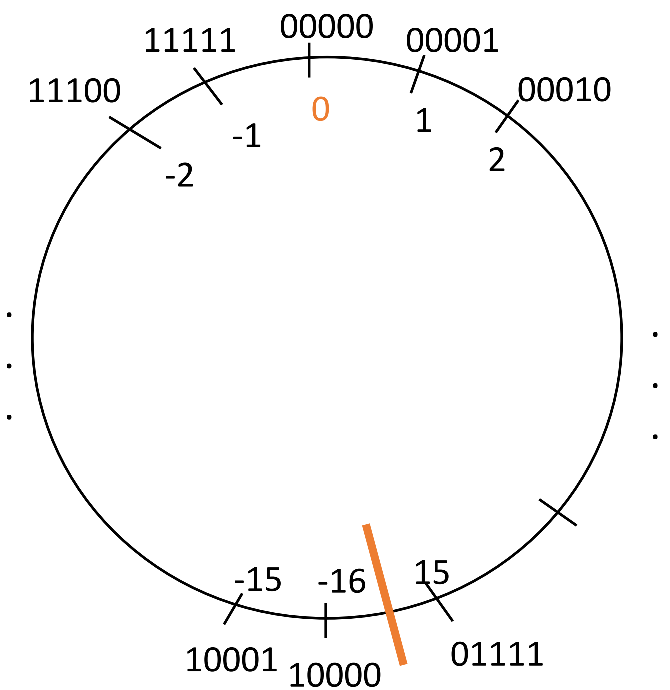
- Observations
  - $|TMin| = TMax + 1$
    - Asymmetric range
  - $UMax = 2 * TMax + 1$

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

	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

Encoding Integers w/ Sign

- Two's Complement
  - e.g., $7_{10} = 00111_2$ $-7_{10} = 11001_2$
  - 2^{w-1} non-negatives (including zero)
  - 2^{w-1} negatives
  - unique zero representation
  - easy for hardware
    - leading 0 : non-negative
    - leading 1 : negative



two's compliment

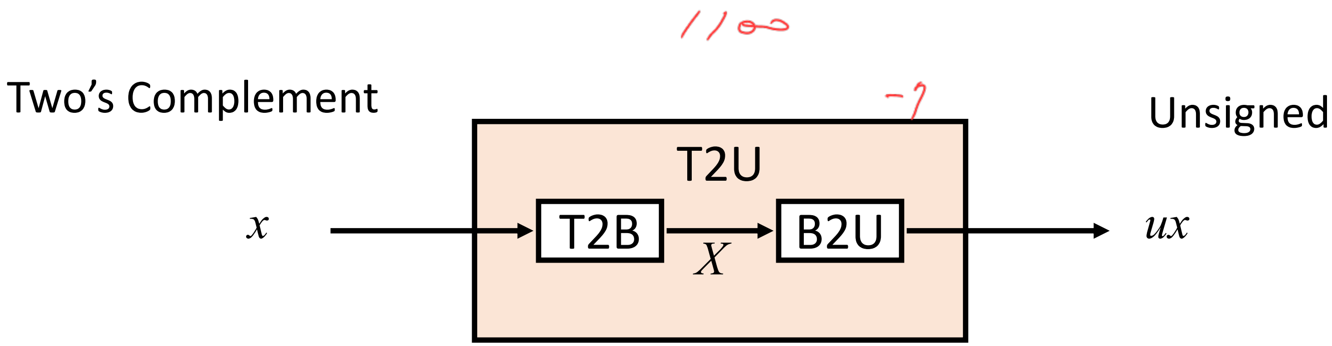
Unsigned & Signed Numeric Values

- Equivalence
  - Same encodings for nonnegative values
- Uniqueness
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding
- $\Rightarrow$ Can Invert mappings
  - U2B($\mathbf{x}$) = B2U^{-1}($\mathbf{x}$)
    - Bit pattern for unsigned integer
  - T2B($\mathbf{x}$) = B2T^{-1}($\mathbf{x}$)
    - Bit pattern for two's complement integer

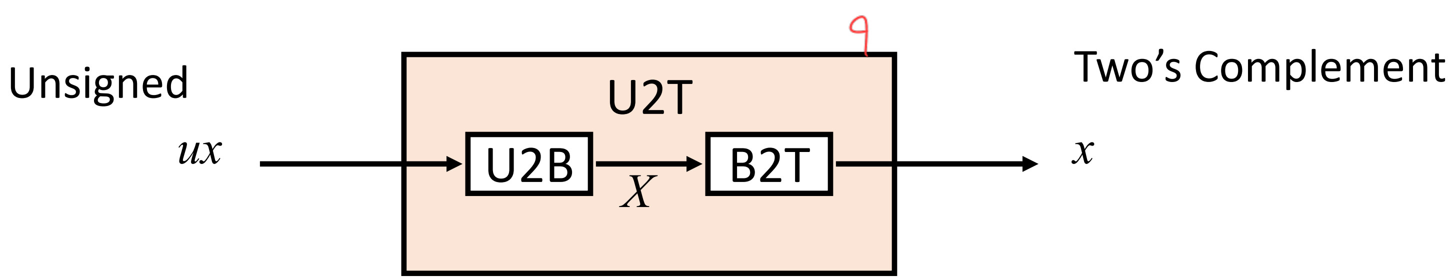
x	B2U(x)	B2T(x)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

Mapping Btwn. Signed & Unsigned

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



Maintain Same Bit Pattern



Maintain Same Bit Pattern

Mapping Signed ↔ Unsigned

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

$\xrightarrow{\text{T2U}}$

$\xleftarrow{\text{U2T}}$

Mapping Signed ↔ Unsigned

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

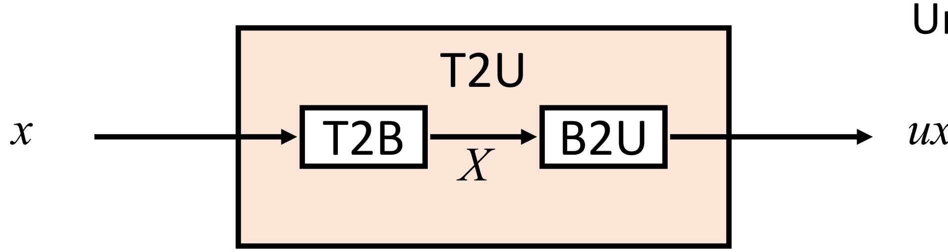
=

+/- 16 (2^w)

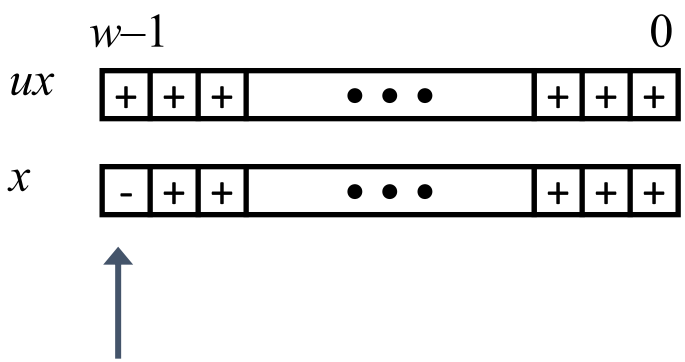
Relation btwn. Signed & Unsigned

Two's Complement

Unsigned



Maintain Same Bit Pattern



Large negative weight

becomes

Large positive weight

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

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

- Two's complement, 4 bits

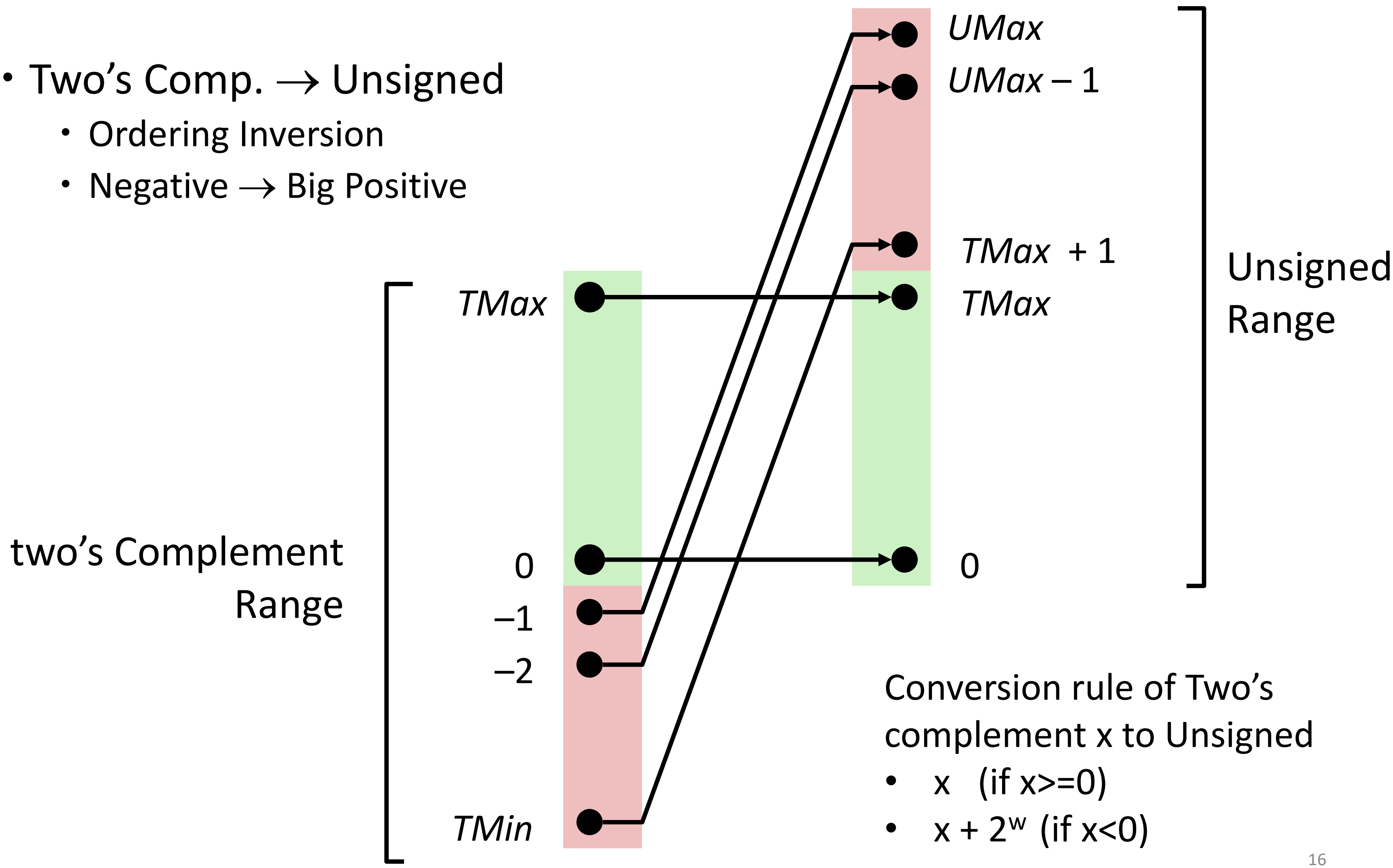
0111 (7, TMax)
:
:
0000 (0)
1111 (-1)
:
:
1000 (-8, TMin)

- Unsigned, 4 bits

1111 (15, UMax)
:
: 1000 (8)
0111 (7)
:
:
0000 (0)

Conversion Visualized

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



Conversion rule of Two's complement x to Unsigned

- x (if x>=0)
- x + 2^w (if x<0)

Signed vs. Unsigned in C programming

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

    `0U, 4294967259U`

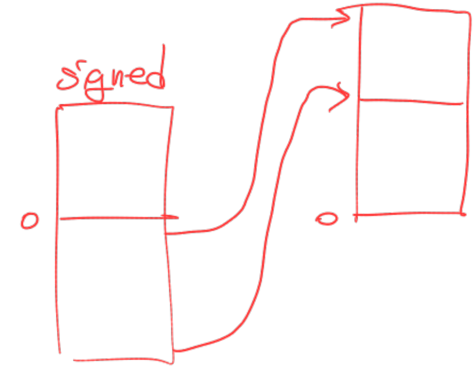
- Casting
  - **Explicit** casting btwn. signed & unsigned same as U2T and T2U

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

  - **Implicit** casting also occurs via assignments and procedure calls

```
tx = ux;
uy = ty;
```

Casting Surprises



- Expression Evaluation
  - If there is a mix of unsigned and signed in single expression, ***signed values implicitly cast to unsigned***
  - Including comparison operations **<, >, ==, <=, >=**
  - Examples (W=32):

  TMIN = -2,147,483,648 , TMAX = 2,147,483,647

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

Summary: Casting Signed ↔ Unsigned

- Signed and unsigned casting
  - **Bit pattern is maintained**
  - **But reinterpreted**
  - Can have unexpected effects: adding or subtracting 2^w

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

Quiz : output ?

- Hint : unsigned i

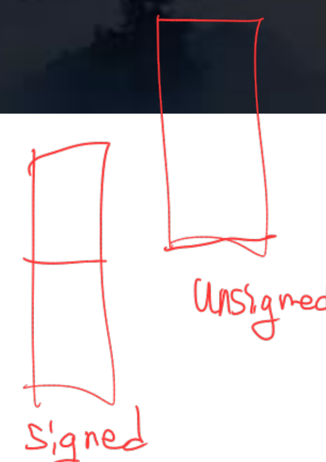
```c
#include <stdio.h>

int main() {
        unsigned  i;
        int a[3] = {1, 2, 3};
        for (i = sizeof(a)/sizeof(int) - 1; i>=0; i--)
                printf("%d\n", a[i]);
        return 0;
}
```

signed

i i i

2 1 0 .

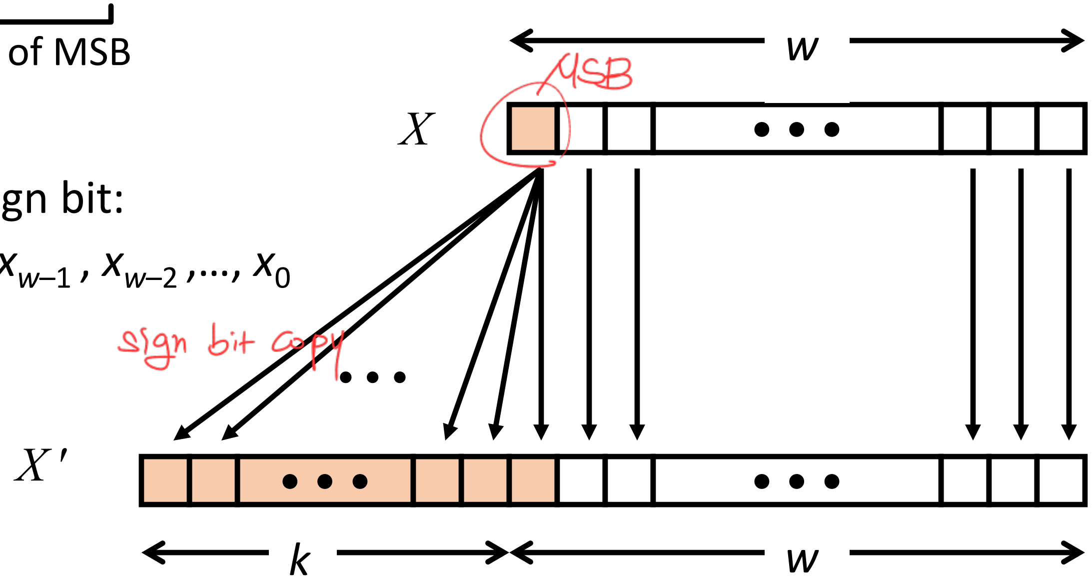


Sign Extension

- Sign Extension int → long
  4byte 8byte
  - *e.g.* 32 bit integer -> 64 bit integer
  - Extend sign bit for the higher bits
- Task
  - Given ***w*-bit signed integer *x***
  - Convert it to ***w+k-bit*** integer with **same value**

  k copies of MSB

- Rule:
  - Make *k* copies of sign bit:
  - $X' = x_{w-1}, \dots, x_{w-1}, x_{w-1}, x_{w-2}, \dots, x_0$



Sign Extension Example

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

```
  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

Sign Extension Example, why

- 1100 : -8 + 4 = -4
- 11100 : -16 + 8 + 4 = -4

Add -2^w : -16 (decrease)

Convert -2^{W-1} to 2^{w-1} : -8 → 8 (increase)

$2^{W-1} * 2 = 2^W$

Truncating Numbers

Truncate : cut

- Truncating a number can alter its value
  - A form of overflow

- Truncating an unsigned x (w bit long) to x' (k bit long)
  - Truncating x to k bits is equivalent to computing x mod 2^k

  $B2U_k([x_{k-1}, x_{k-2}, \ldots, x_0]) = \mathbf{B2U_w([x_{w-1}, x_{w-2}, \ldots, x_0]) \bmod 2^k}$

- Truncating a signed x (w bit long) to x' (k bit long)

  $B2T_k([x_{k-1}, x_{k-2}, \ldots, x_0]) = U2T_k(\mathbf{B2U_w([x_{w-1}, x_{w-2}, \ldots, x_0]) \bmod 2^k})$

4byte

2byte

```
int         x =  50323;          // 0x0000C493
short int   sx = (short) x;      // -15213
int          y = sx;             // -15213
```

0xC493

Summary - Expanding, Truncating

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

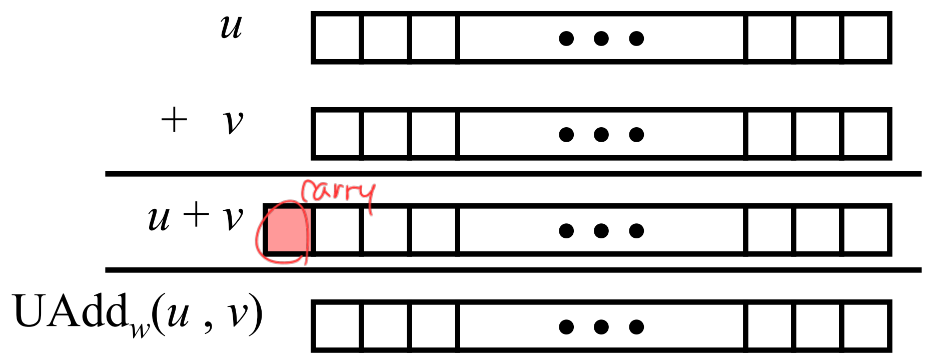
- Truncating (e.g., unsigned to unsigned short)
  - Unsigned/signed: bits are truncated
  - Result reinterpreted
  - Unsigned: mod operation
  - Signed: similar to mod
  - For small numbers yields expected behavior

Unsigned Addition

Operands: *w* bits

True Sum: *w*+1 bits

Discard Carry: *w* bits



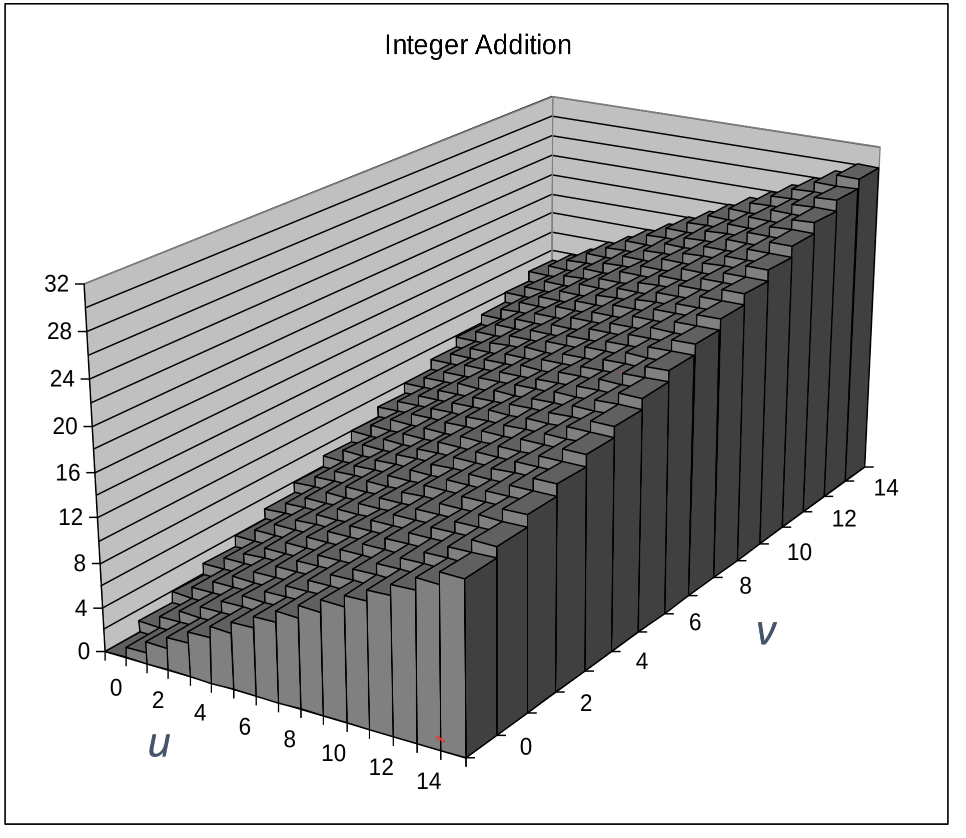
- Standard Addition Function
  - Ignores carry output
- Implements Modular Arithmetic

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

Visualizing (Mathematical) Integer Addition

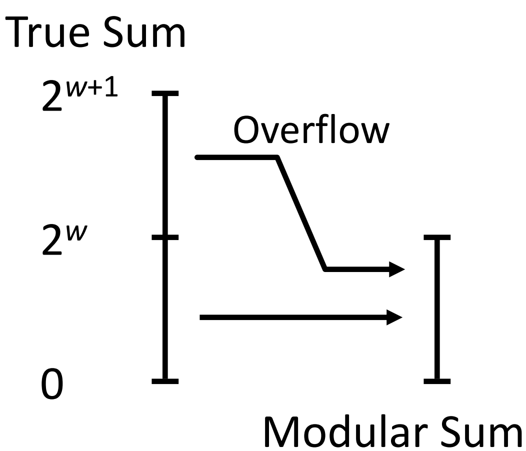
- Integer Addition
  - 4-bit integers *u*, *v*
  - Compute true sum $\text{Add}_4(u\ ,\ v)$
  - Values increase linearly with *u* and *v*

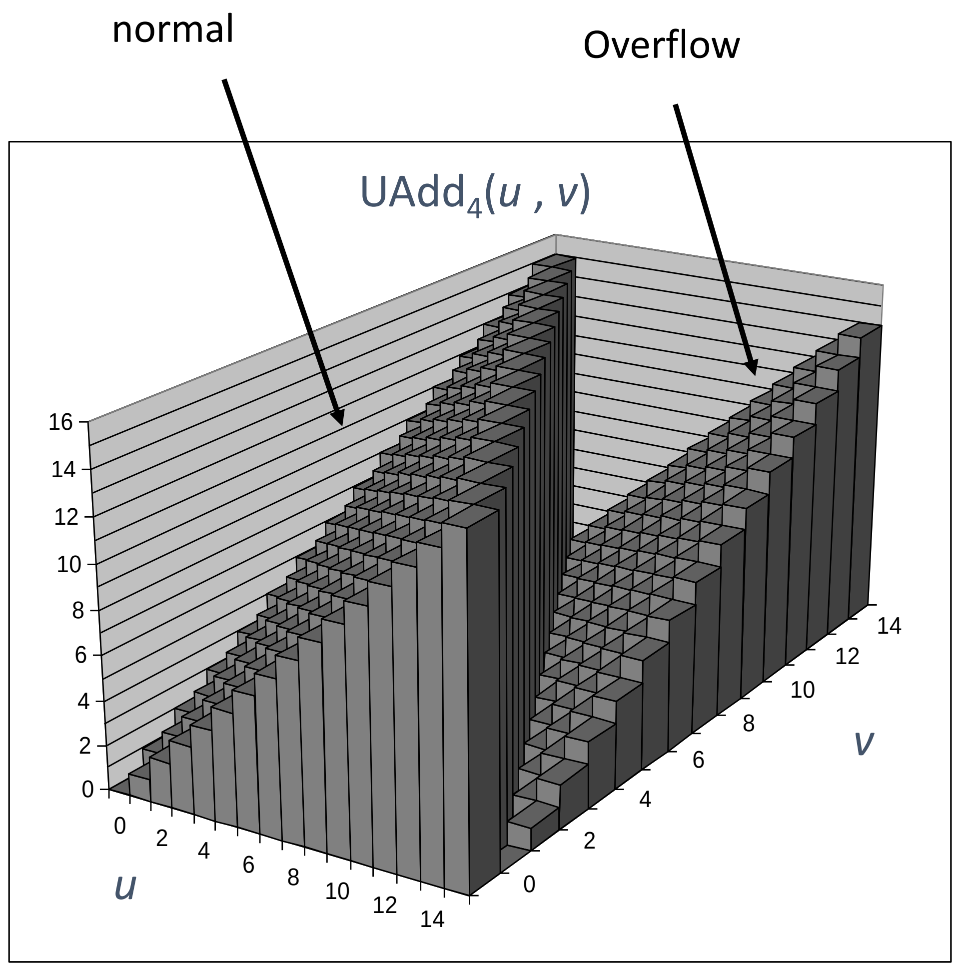
$\text{Add}_4(u\ ,\ v)$



Visualizing Unsigned Addition

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





Two's Complement Addition

- **TAdd and UAdd have Identical Bit-Level Behavior** addition은 같음 다만 decode에서 해석 갈림
  - Signed vs. unsigned addition in C:

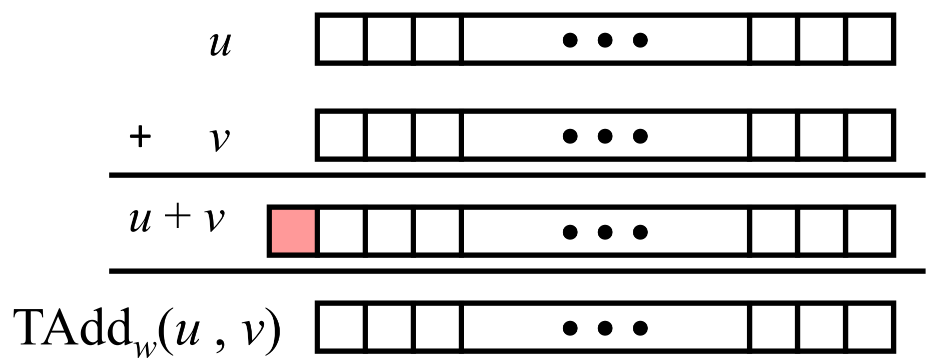
```
int s, t, u, v;
s = (int) ((unsigned) u + (unsigned) v);
t = u + v
```

  - Will give `s == t`

Operands: *w* bits

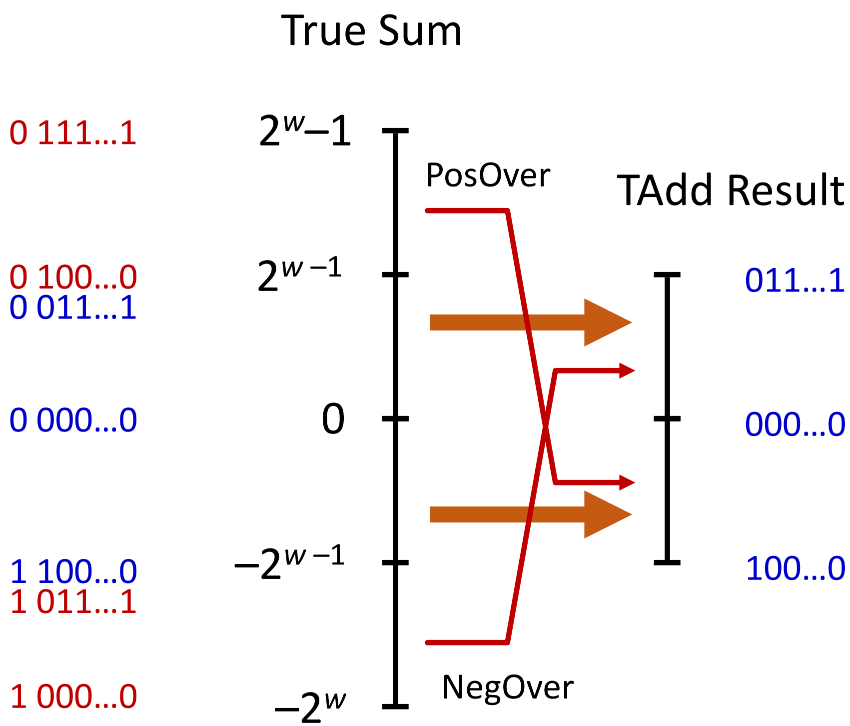
True Sum: *w*+1 bits

Discard Carry: *w* bits



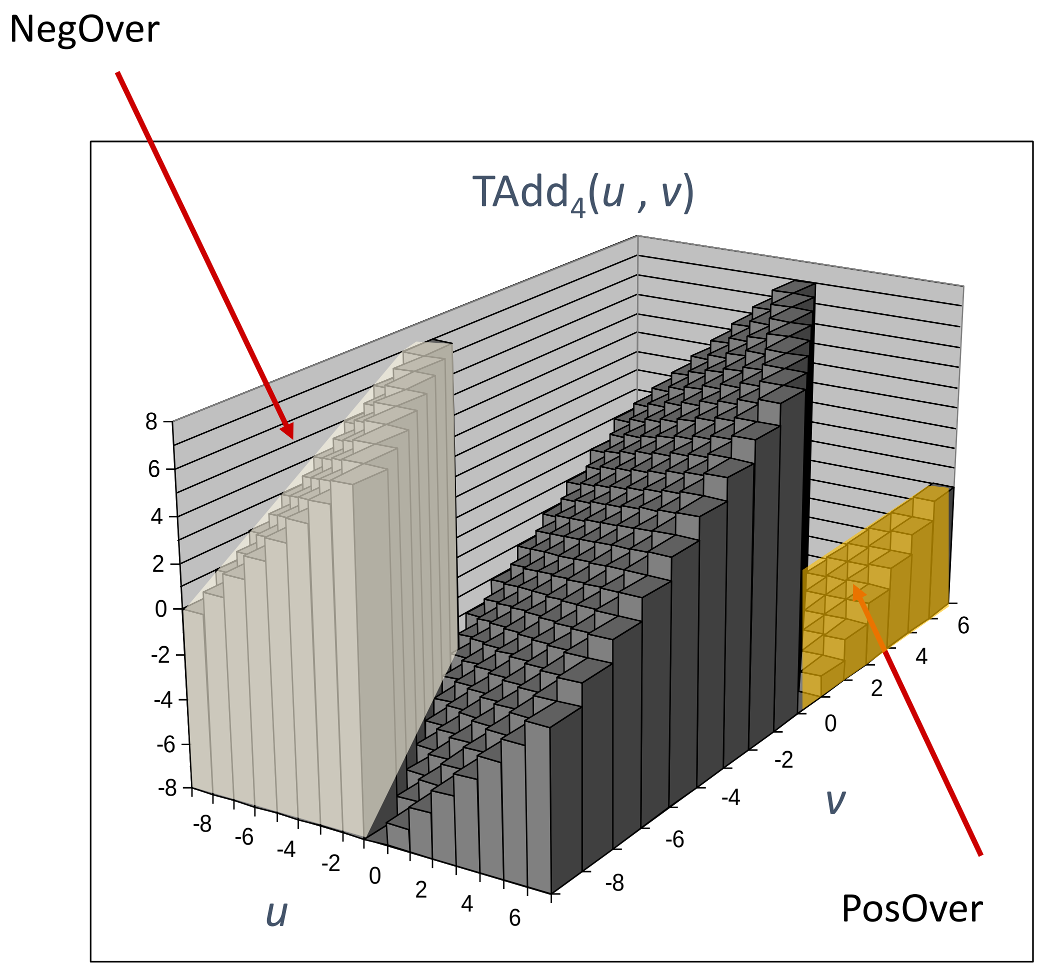
TAdd Overflow

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



Visualizing 2’s Complement Addition

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



Unsigned Multiplication in C

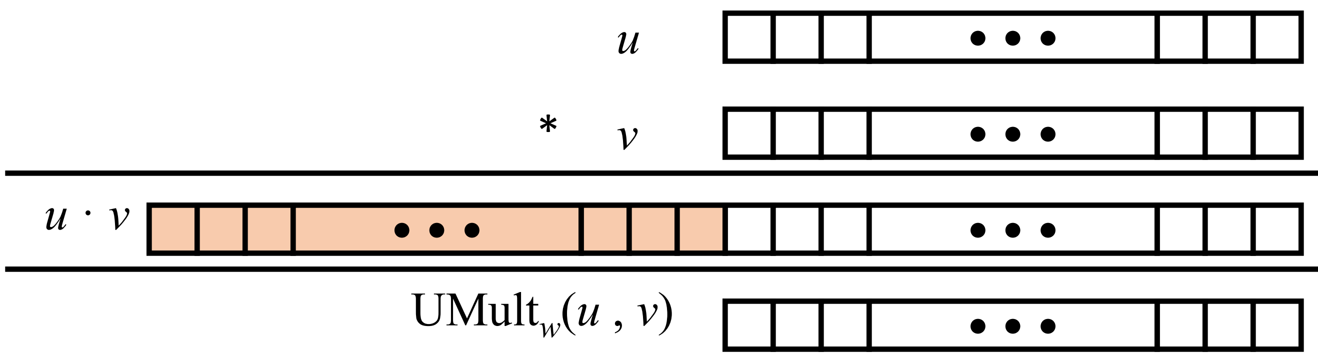
- Standard Multiplication Function
  - **Ignores high order *w* bits**
- Implements Modular Arithmetic

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

Operands: *w* bits

True Product: 2**w* bits

Discard *w* bits: *w* bits



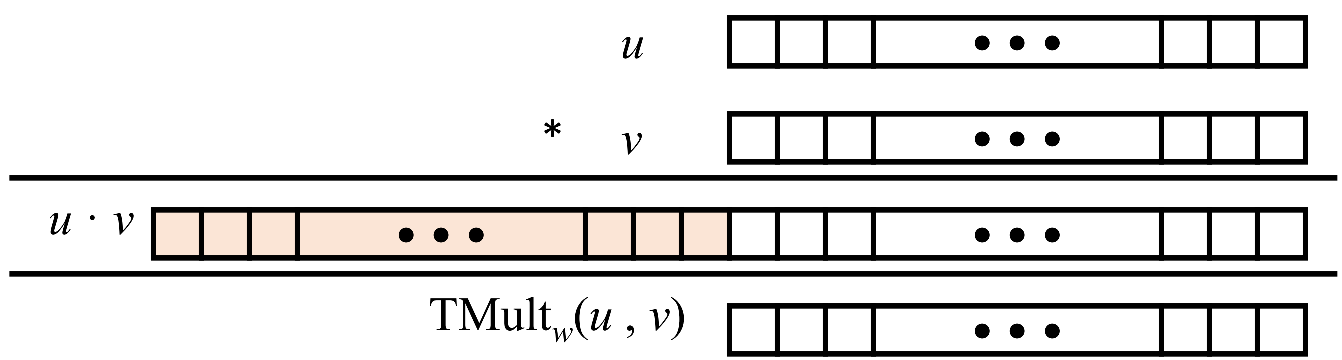
Signed Multiplication in C

- Standard Multiplication Function
  - Ignores high order *w* bits after multiplication

Operands: *w* bits

True Product: 2**w* bits

Discard *w* bits: *w* bits



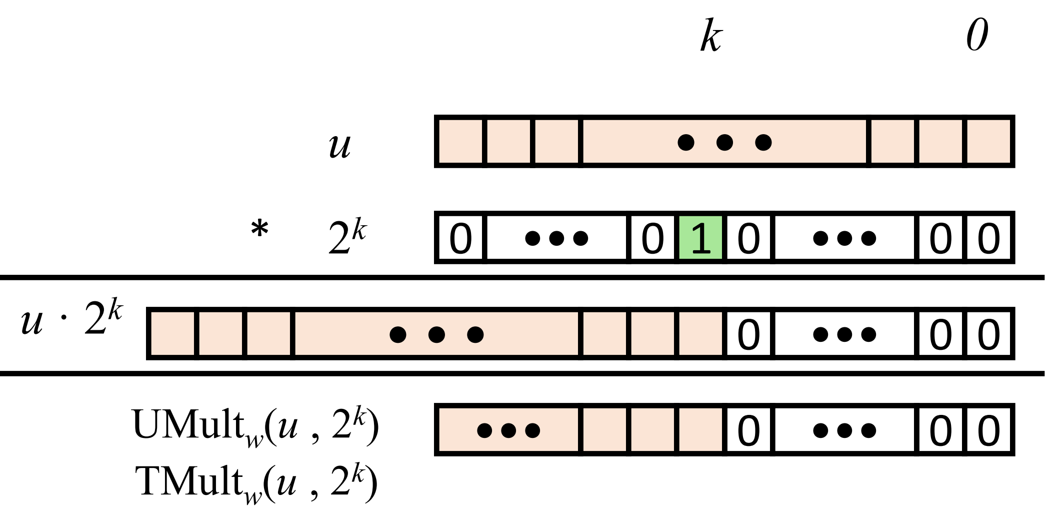
Power-of-2 Multiply with Shift

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

  Operands: *w* bits

  True Product: *w*+*k* bits

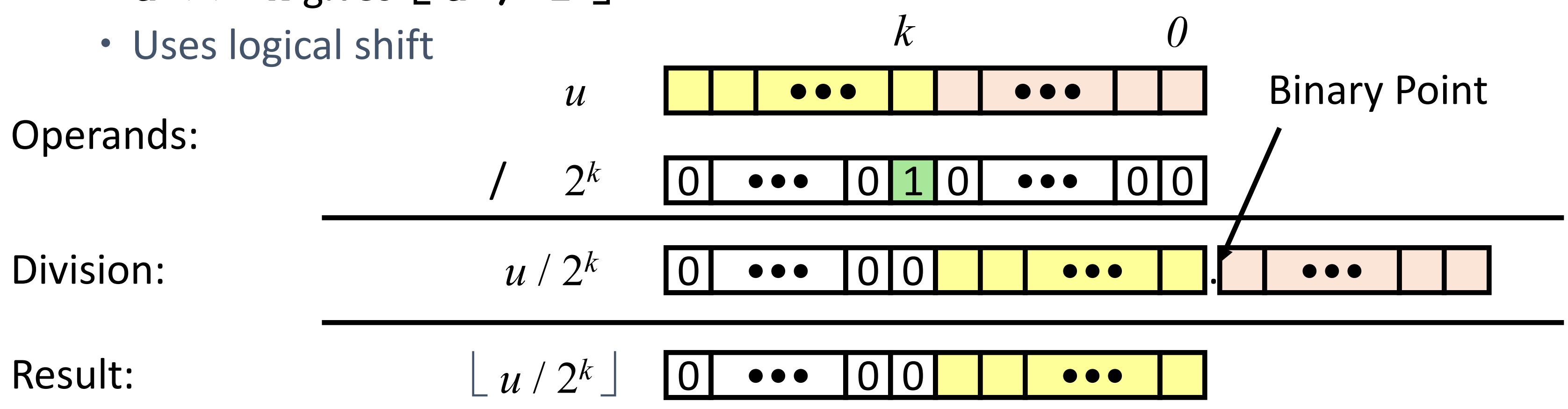
  Discard *k* bits: *w* bits



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

Unsigned Power-of-2 Divide with Shift

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



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

Signed Power-of-2 Mult./Div.

- Two's complement **multiplication** by a power of 2
  - int s, unsigned k $(0 \le k < w)$
  - s << k (in C program) yields $s *^{t}_{w} 2^k$

- Two's complement **division** by a power of 2
  - int s, unsigned k $(0 \le k < w)$
  - s >> k (in C program) yields $\lfloor s / 2^k \rfloor$
  - (NOTE) >> should be an **arithmetic shift-right**

- Bit level behaviors for signed and unsigned power-of 2 multiplications/divisions are the same

Data repre.

- Bit pattern
- Int
- Floating point numbers