CMPT 295

Unit - Data Representation

Lecture 3 – Representing integral numbers in memory - unsigned and signed

Last Lecture

- Von Neumann architecture
  - Architecture of most computers
  - Its components: CPU, memory, input and ouput, bus
  - One of its characteristics: Data and code (programs) both stored in memory
- A look at memory: defined *byte-addressable* memory, diagram of (compressed) memory
  - Word size (w): size of a series of bits (or bit vector) we manipulate, also size of machine words (see Section 2.1.2)
- A look at bits in memory
  - Why binary numeral system (0 and 1 -> two values) is used to represent information in memory
  - Algorithm for converting binary to hexadecimal (hex)
    1. Partition bit vector into groups of 4 bits, starting from right, i.e., least significant byte (LSB)
       - If most significant "byte" (MSB) does not have 8 bits, pad it: add 0's to its left
    2. Translate each group of 4 bits into its hex value
  - What do bits represent? Encoding scheme gives meaning to bits
  - Order of bytes in memory: little endian versus big endian
- Bit manipulation – regardless of what bit vectors represent
  - Boolean algebra: bitwise operations => **AND** (&), **OR** (|), **XOR** (^), **NOT** (~)
  - Shift operations: left shift, right logical shift and right arithmetic shift
    - Logical shift: Fill x with y 0's on left
    - Arithmetic shift: Fill **x** with **y** copies of **x**'s sign bit on left
    - Sign bit: Most significant bit (MSb) before shifting occurred

NOTE:
C logical operators and C bitwise (bit-level) operators behave differently!
Watch out for `&&` versus `&`, `||` versus `|`, …

Today’s Menu

- Representing data in memory – Most of this is review
  - “Under the Hood” - Von Neumann architecture
  - Bits and bytes in memory
    - How to diagram memory -> Used in this course and other references
    - How to represent series of bits -> In binary, in hexadecimal (conversion)
    - What kind of information (data) do series of bits represent -> Encoding scheme
    - Order of bytes in memory -> Endian
  - Bit manipulation – bitwise operations
    - Boolean algebra + Shifting
- Representing integral numbers in memory
  - Unsigned and signed
  - Converting, expanding and truncating
  - Arithmetic operations
- Representing real numbers in memory
  - IEEE floating point representation
  - Floating point in C – casting, rounding, addition, …

Warm up exercise!

As a warm up exercise, fill in the blanks!

- If the context is C (on our *target machine*)
  - `char   =>` _____ bits/ _____ byte
  - `short  =>` _____ bits/ _____ bytes
  - `int    =>` _____ bits/ _____ bytes
  - `long   =>` _____ bits/ _____ bytes
  - `float  =>` _____ bits/ _____ bytes
  - `double=>` _____ bits/ _____ bytes
  - `pointer (e.g. char *)`

    `=>` _____ bits/ _____ bytes

Remember:

Address	M[]
size-1	
⋮	
0x0003	01000010
0x0002	01101001
0x0001	01110100
0x0000	01110011

Unsigned integral numbers

- What if the byte at M[0x0002] represented an unsigned integral number, what would be its value?
- $X = 01101001_2$ $\quad w = 8$

  A series of bits => bit vector

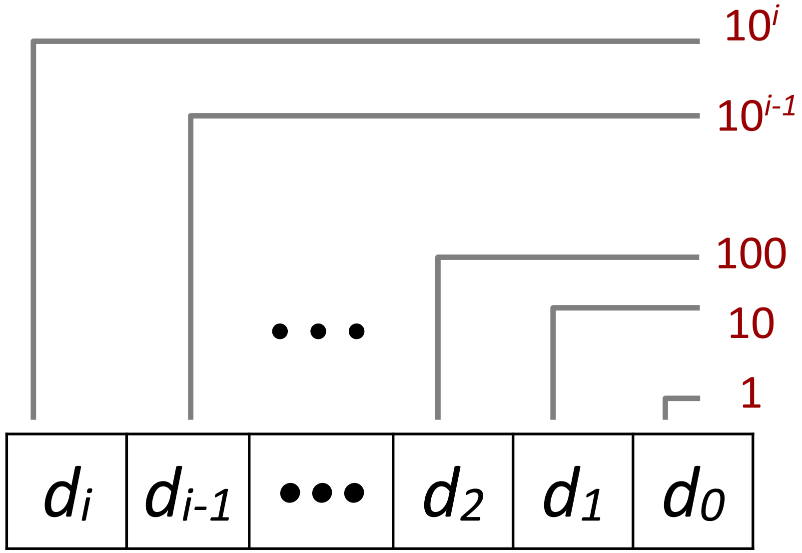
  w =>width of the bit vector
- Let's apply the encoding scheme: $B2U(X) = \sum_{i=0}^{w-1} x_i \cdot 2^i$

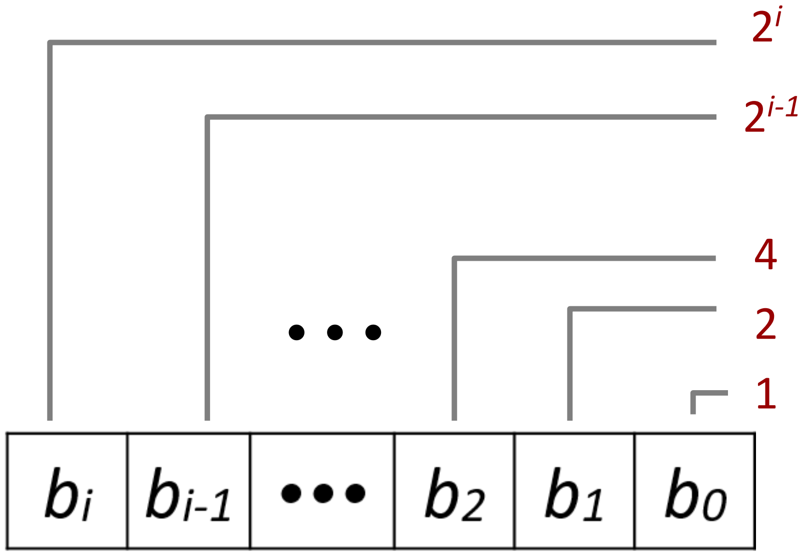
$0 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 =$

- For $w = 8$, range of possible unsigned values: [ ]
- For any w, range of possible unsigned values: [ ]
- Conclusion: w bits can only represent a fixed # of possible values, but these w bits represent these values **exactly**

$B2U(X)$ Conversion (Encoding scheme)

- Positional notation: expand and sum all terms





Example: $246_{10} = 2 \times 10^2 + 4 \times 10^1 + 6 \times 10^0$

1's $= 10^0$
10's $= 10^1$
100's $= 10^2$

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

Range of possible values?

- If the context is C (on our *target machine*)
  - `unsigned char`?
  - `unsigned short`?
  - `unsigned int`?
  - `unsigned long`?

Examples of "Show your work"

$U2B(X)$ Conversion (into 8-bit binary # => w = 8)

Method 1 - Using subtraction:
subtracting decreasing power of 2 until reach 0

246 =>	246 – 128 = 118	->	$128 = 1 \times 2^7$
	118 – 64 = 54	->	$64 = 1 \times 2^6$
	54 – 32 = 22	->	$32 = 1 \times 2^5$
	22 – 16 = 6	->	$16 = 1 \times 2^4$
	6 – 8 = nop!	->	$8 = 0 \times 2^3$
	6 – 4 = 2	->	$4 = 1 \times 2^2$
	2 – 2 = 0	->	$2 = 1 \times 2^1$
	0 – 1 = nop!	->	$1 = 0 \times 2^0$

246 => $1\ 1\ 1\ 1\ 0\ 1\ 1\ 0_2$

Method 2 - Using division:
dividing by 2 until reach 0

246 =>	246 / 2 = 123	->	R = 0
	123 / 2 = 61	->	R = 1
	61 / 2 = 30	->	R = 1
	30 / 2 = 15	->	R = 0
	15 / 2 = 7	->	R = 1
	7 / 2 = 3	->	R = 1
	3 / 2 = 1	->	R = 1
	1 / 2 = 0	->	R = 1

246 => $1\ 1\ 1\ 1\ 0\ 1\ 1\ 0_2$

$U2B(X)$ Conversion – A few tricks

- Decimal -> binary
  - Trick: When decimal number is 2^n, then its binary representation is 1 followed by n zero's
  - Let's try: if $X = 32 \Rightarrow X = 2^5$, then n = 5 => 10000_2 $(w = 5)$

    What if $w = 8$?

    Check: $1 \times 2^4 = 32$
- Decimal -> hex
  - Trick: When decimal number is 2^n, then its hexadecimal representation is 2^i followed by j zero's, where n = i + 4j and 0 <= i <=3
  - Let try: if $X = 8192 \Rightarrow X = 2^{13}$, then n = 13 and 13 = i + 4j => 1 + 4 x 3

    => 0x2000

    Convert 0x2000 into a binary number:

    Check: $2 \times 16^3 = 2 \times 4096 = 8192$

Signed integral numbers

Remember:

Address	M[]
size-1	
⋮	
0x0003	01000010
0x0002	01101001
0x0001	01110100
0x0000	01110011

- What if the byte at M[0x0001] represented a signed integral number, what would be its value?
- $X = 11110100_2 \quad w = 8$
- Let's apply the encoding scheme:

T => Two's Complement

w =>width of the bit vector

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

$-1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 =$

- What would be the bit pattern of the …
  - Most negative value:
  - Most positive value:
- For $w = 8$, range of possible signed values: [ ]
- For any w, range of possible signed values: [ ]
- Conclusion: same as for unsigned integral numbers

Examples of “Show your work”

$T2B(X)$ Conversion -> Two’s Complement

$w = 8$

Method 1 If $X < 0$, **(~(U2B(|X|)))+1**

If X = -14 (and 8 bit binary #s)

1. |X| **=>** |-14| =
2. **U2B(**14**) =>**
3. **~(**00001110_2**) =>**
4. **(**11110001_2**)+1 =>**

Binary addition:

```
  11110001
+ 00000001
```

Method 2 If $X < 0$, **U2B(X + 2^w)**

If X = -14 (and 8 bit binary #s)

1. X **+ 2^w =>** -14 +
2. **U2B(**242**) =>**

Using subtraction:

242 – 128 = 114 -> 1 x 2^7
114 – 64 = 50 -> 1 x 2^6
50 – 32 = 18 -> 1 x 2^5
18 – 16 = 2 -> 1 x 2^4
2 – 8 -> nop! -> 0 x 2^3
2 – 4 -> nop! -> 0 x 2^2
2 – 2 = 0 -> 1 x 2^1
0 – 1 -> nop! -> 0 x 2^0

Check:

Properties of unsigned & signed conversions

$w = 4$

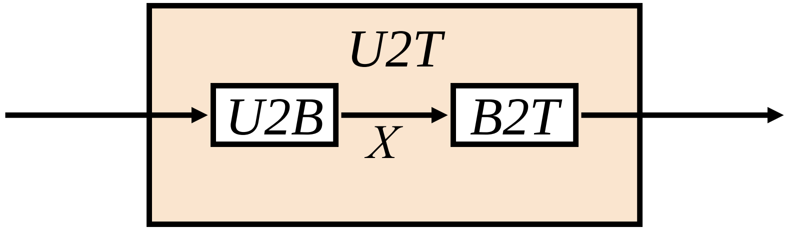
X	$B2U(X)$	$B2T(X)$
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

- Equivalence
  - Both encoding schemes ($B2U$ and $B2T$) produce the same bit patterns for nonnegative values
- Uniqueness
  - Every bit pattern produced by these encoding schemes ($B2U$ and $B2T$) represents a unique (and exact) integer value
  - Each representable integer has unique bit pattern

Converting between signed & unsigned
of same size (same data type)

$w = 8$

Unsigned ux



Signed (Two's Complement) x

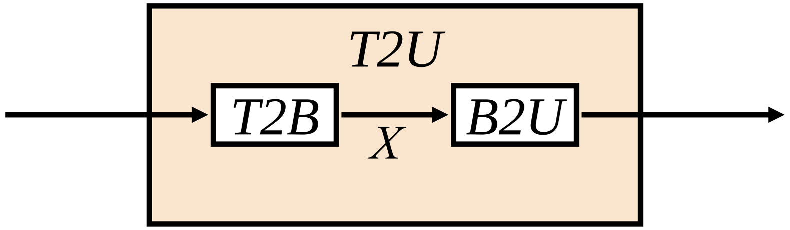
If $ux = 129_{10}$

Maintain Same Bit Pattern

then $x =$

$w = 4$

Signed (Two's Complement) x



Unsigned ux

If $x = -5_{10}$

Maintain Same Bit Pattern

then $ux =$

- Conclusion - Converting between unsigned and signed numbers: Both have same bit pattern, however, this bit pattern may be interpreted differently, i.e., producing a different value

Converting signed ↔ unsigned with $w = 4$

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

$T2U(X)$

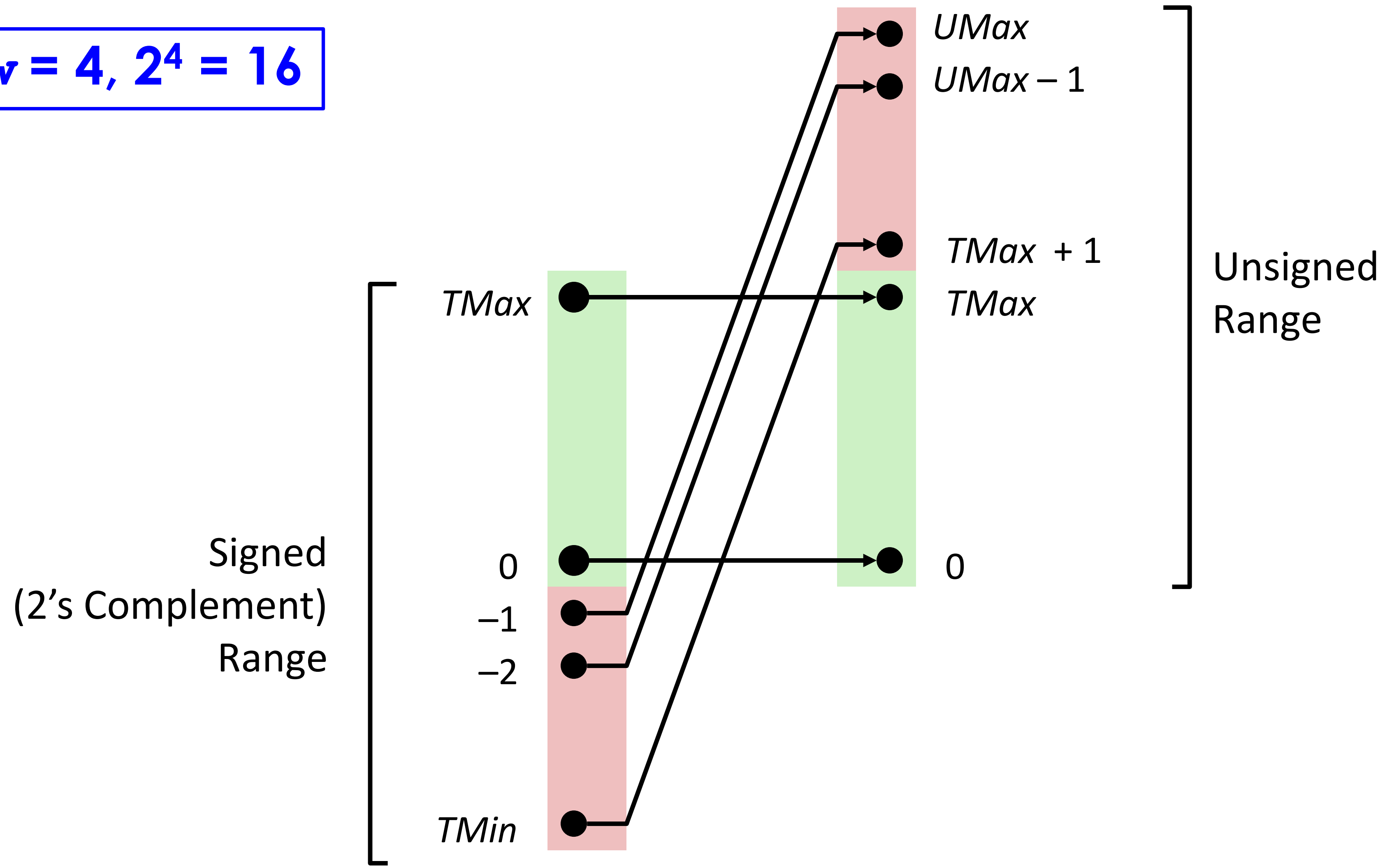
$U2T(X)$

$+ 16\ (+2^4)$ $\quad T2U(X)$

$- 16\ (+2^4)$ $\quad U2T(X)$

Visualizing the relationship between signed & unsigned

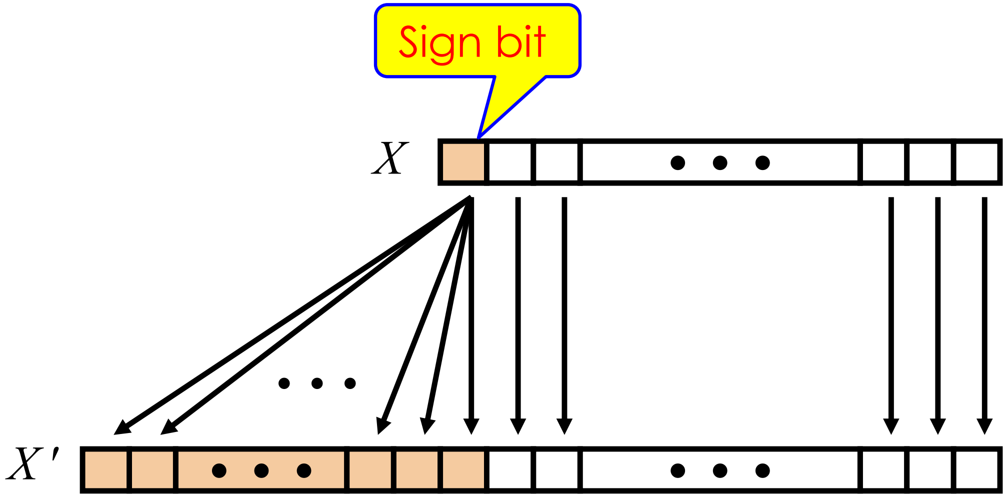
If $w = 4$, $2^4 = 16$



Sign extension

- Converting unsigned (or signed) of different sizes (different data types)

1. Small data type -> larger
   - Sign extension
     - Unsigned: zero extension
     - Signed: sign bit extension



- Conclusion: Value unchanged
- Let's try:
  - Going from a data type that has a width of 3 bits (w = 3) to a data type that has a width of 5 bits (w = 5)

- Unsigned: X = 3 => 011_2 w = 3 X = 4 => 100_2 w = 3

  new X = <= w = 5   new X = <= w = 5

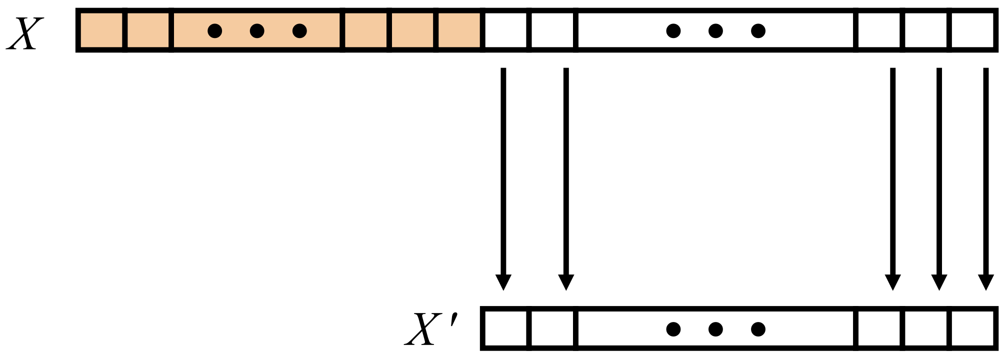
- Signed: X = 3 => 011_2 w = 3 X = -3 => 101_2 w = 3

  new X = <= w = 5   new X = <= w = 5

Truncation

- Converting unsigned (or signed) of different sizes(different data types)

2. Large data type -> smaller
   - Truncation



- Conclusion: Value may be altered
  - A form of overflow
- Let's try:
  - Going from a data type that has a width of 5 bits (w = 5) to a data type that has a width of 3 bits (w = 3)
- Unsigned: X = 27 => 11011_2 $w = 5$

  new X = <= $w = 3$
- Signed: X = -15 => 10001_2 $w = 5$ X = -1 => 11111_2 $w = 5$

  new X = <= $w = 3$ new X = <= $w = 3$

Summary

- Interpretation of bit pattern B into either unsigned value U or signed value T
  - *B2U(X)* and *U2B(X)* encoding schemes (conversion)
  - *B2T(X)* and *T2B(X)* encoding schemes (conversion)
    - Signed value expressed as two's complement => *T*
- Conversions from unsigned <-> signed values
  - *U2T(X)* and *T2U(X)* => adding or subtracting 2^w
- Implication in C: when converting (implicitly via promotion and explicitly via casting):
  - Sign:
    - Unsigned <-> signed (of same size) -> Both have same bit pattern, however, this bit pattern may be interpreted differently
      - Can have unexpected effects -> producing a different value
  - Size:
    - Small -> large (for signed, e.g., `short` to `int` and for unsigned, e.g., `unsigned short` to `unsigned int`)
      - sign extension: For unsigned -> zeros extension, for signed -> sign bit extension
      - Both yield expected result –> resulting value unchanged
    - Large -> small (e.g., `unsigned int` to `unsigned short`)
      - truncation: Unsigned/signed -> most significant bits are truncated (discarded)
      - May not yield expected results -> original value may be altered
  - Both (sign and size): 1) **size** conversion is first done then 2) **sign** conversion is done

Next Lecture

- Representing data in memory – Most of this is review
  - "Under the Hood" - Von Neumann architecture
  - Bits and bytes in memory
    - How to diagram memory -> Used in this course and other references
    - How to represent series of bits -> In binary, in hexadecimal (conversion)
    - What kind of information (data) do series of bits represent -> Encoding scheme
    - Order of bytes in memory -> Endian
  - Bit manipulation – bitwise operations
    - Boolean algebra + Shifting
- Representing integral numbers in memory
  - Unsigned and signed
  - Converting, expanding and truncating
  - Arithmetic operations
- Representing real numbers in memory
  - IEEE floating point representation
  - Floating point in C – casting, rounding, addition, …