Integers

Computer Systems Organization

Today

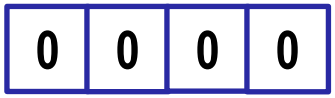
Integers

- Representation: unsigned and signed
- Conversion, casting
- Expanding

How are **Unsigned Integers** Represented in a Computer?

Unsigned Numbers

- An unsigned number is represented as a sequence of bits
- 0 is represented as



- For the following numbers, add 0001
- With 4 bit, 16 values can be represented (i.e., 0 to 15)

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

Binary <-> Unsigned Integers

- **Translate a binary to an unsigned number**

256	128	64	32	16	8	4	2	1	2^i
0	0	1	0	0	1	0	1	1	x_i

1*1 + 1*2 + 0*4 + 1*8 + 0*16 + 0*32 + 1*64 + 0*128 + 0*256 = 75

- **Binary to Unsigned**

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

How are **Signed Integers** Represented in a Computer?

Binary <-> Signed Integers

- **Translate a binary to a signed number**

Sign bit

256	128	64	32	16	8	4	2	1
1	0	1	0	0	1	0	1	1

- 1*256

1*1 + 1*2 + 0*4 + 1*8 + 0*16 + 0*32 + 1*64 + 0*128 - 1*256 = -181

- This is called the **two's complement** representation
- **Binary to two's complement**

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

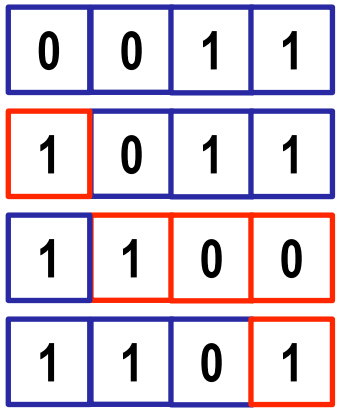
Sign Bit

Two's Complement

- To calculate the two's complement of a positive number
  - Set the most significant bit to 1
  - Flip all the other bits
  - Add 1

Two's Complement

- Example: calculate the two's complement of +3

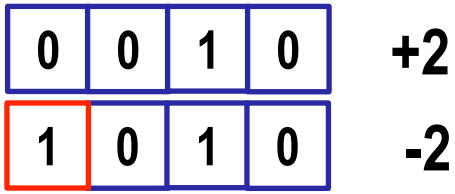


Why Two's Complement?

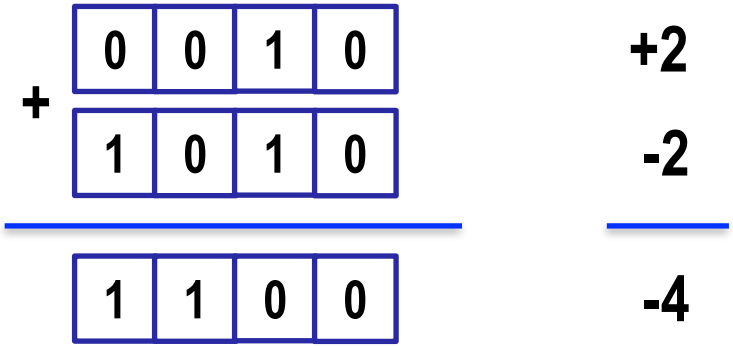
- Another representation: the signed magnitude
- The most significant bit indicates whether a number is positive or negative

Signed Magnitude

- Example: -2 is represented as follows

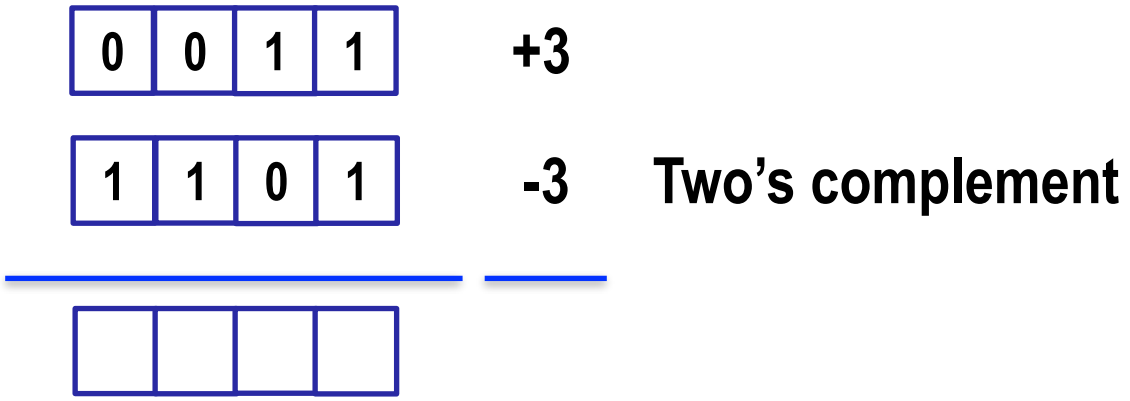


- Let's see the result of adding +2 and -2

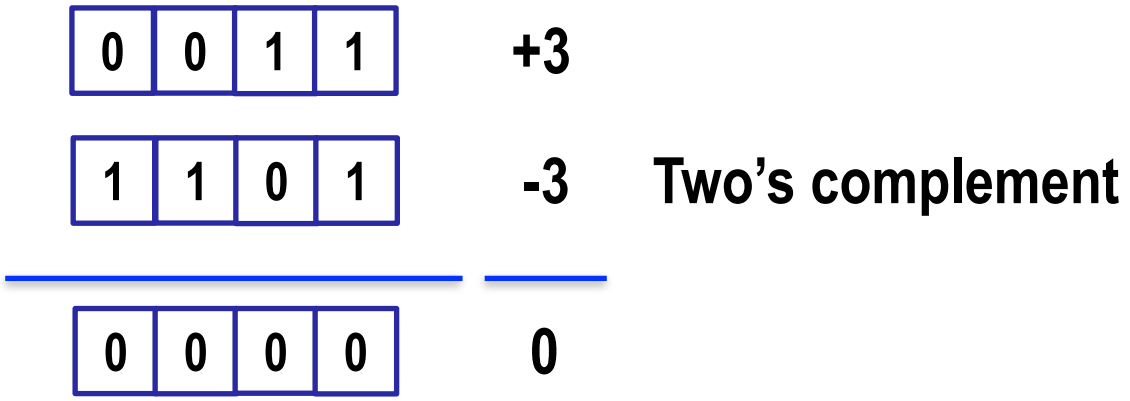


- The sum is not 0

Two’s Complement



Two's Complement



- The sum is 0
- Two's complement avoids the previous problem of signed magnitude
- More details: https://www.youtube.com/watch?v=Z3mswCN2FJs

Encoding Integers

Unsigned

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

Two's Complement

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

```
short int x =  15213;
short int y = -15213;
```

Sign Bit

- **C `short` 2 bytes long**

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

- **Sign Bit**
  - For 2's complement, most significant bit indicates sign
    - 0 for nonnegative
    - 1 for negative

Numeric Ranges

- **Unsigned Values**
  - *UMin* = 0

    000…0
  - *UMax* = $2^w - 1$

    111…1

- **What is the range of two's complement values?**
- **Two's Complement Values**
  - *TMin* = -2^{w-1}

    100…0
  - *TMax* = $2^{w-1} - 1$

    011…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

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

Unsigned & Signed Numeric Values

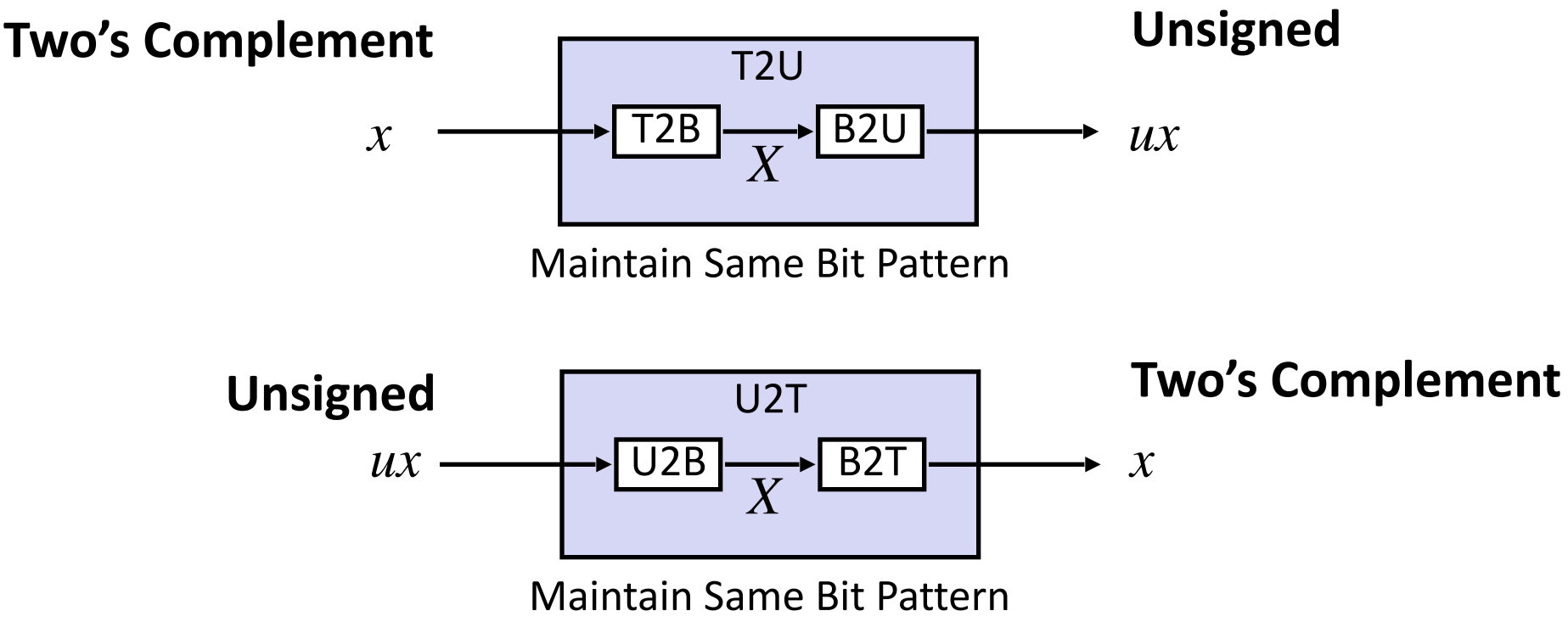
X	B2U(*X*)	B2T(*X*)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

- **Equivalence**
  - Same encodings for nonnegative values
- **Uniqueness**
  - Every bit pattern represents unique integer value
  - Each representable integer has unique bit encoding

Today: Bits, Bytes, and Integers

- **Integers**
  - Representation: unsigned and signed
  - **Conversion, casting**
  - Expanding

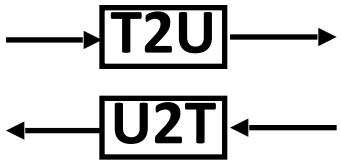
Mapping Between Signed & Unsigned



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

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



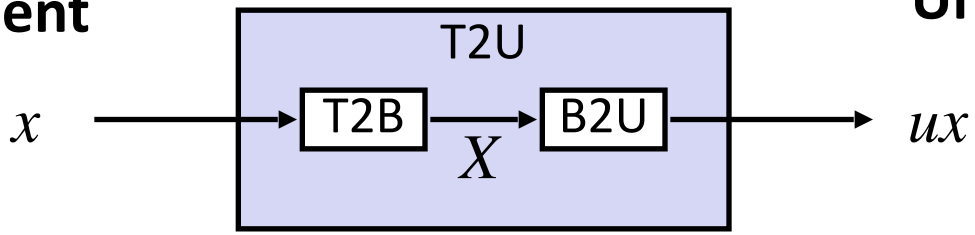
Mapping Signed ↔ Unsigned

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	+/- 16	10
1011	-5	⟷	11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

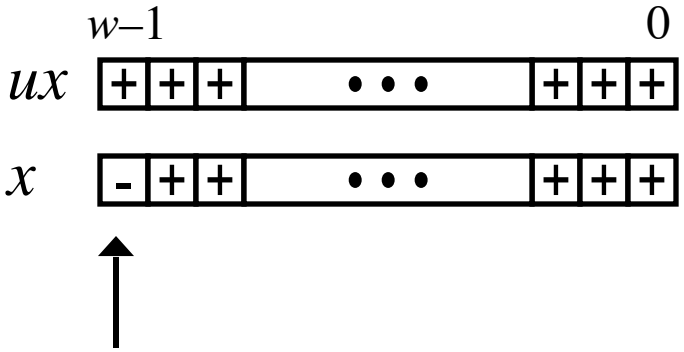
Relation between Signed & Unsigned

Two's Complement

Unsigned



Maintain Same Bit Pattern



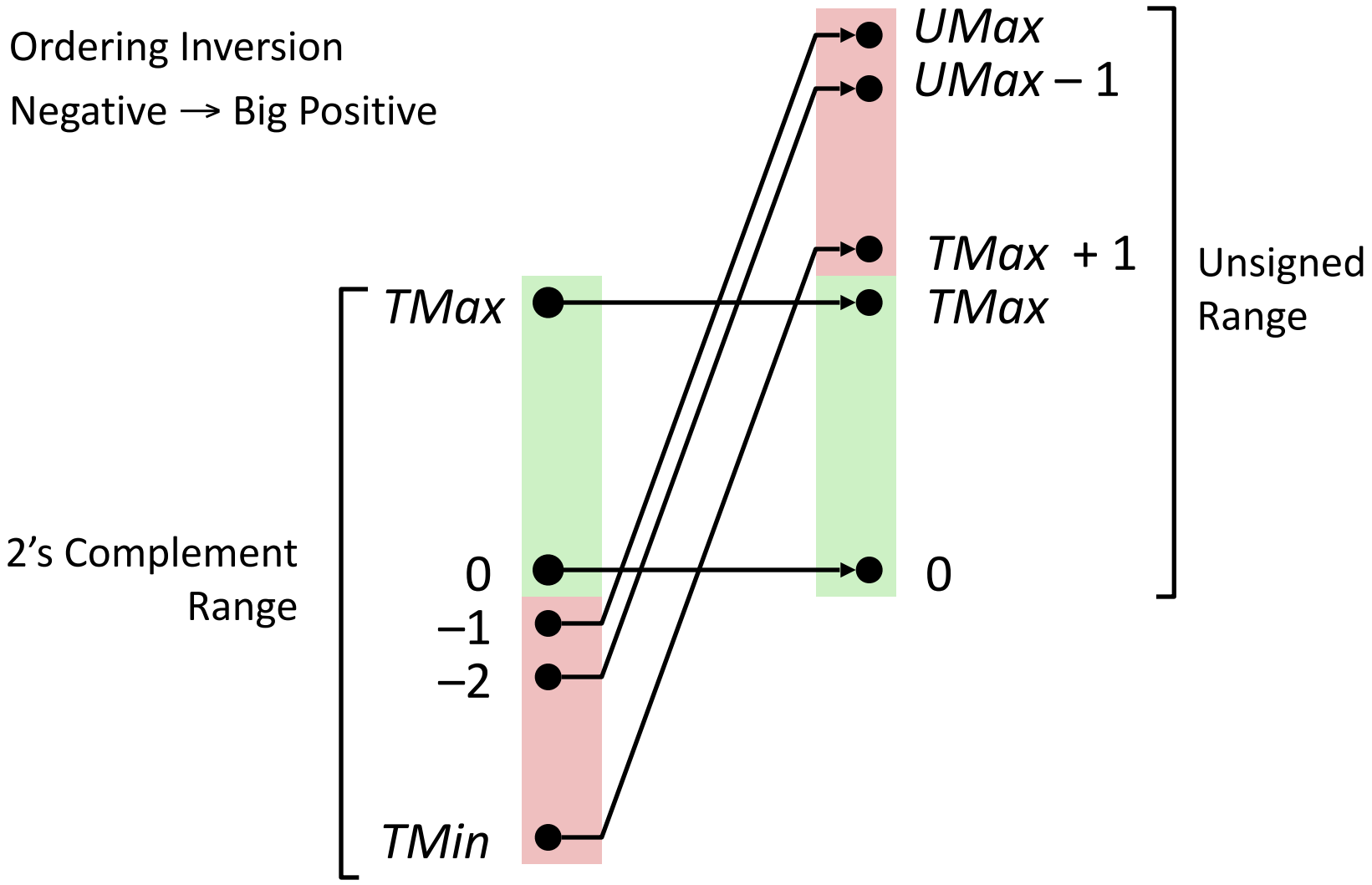
Large negative weight

becomes

Large positive weight

Conversion Visualized

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



Demo Code (loop_unsigned_iterator.c)

```
int main(void)
{
    unsigned i;

    for (i = 10; i >= 0; i--)
            printf(" i = %u \n", i);

    return 0;
}
```

What would be the result?

Signed vs. Unsigned in C

Constants

- By default are considered to be signed integers
- Unsigned if have “U” as suffix

```
0U, 4294967259U
```

Casting

- Explicit casting between signed & unsigned same

```
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 **<, >, ==, <=, >=**

Constant$_1$	**Constant$_2$**	**Relation**	**Evaluation**
0	0U		
-1	0		
-1	0U		
2147483647	-2147483647-1		

```
int main(void)
{
    unsigned i;

    for (i = 10; i >= 0; i--)
            printf(" i = %u \n", i);

    return 0;
}
```

When i reaches 0 and is decremented, it becomes -1 which is interpreted as UMAX since i is unsigned

Find the bug in this code

```
float sum_elements(float a[], unsigned length)
{
  int i;
  float result = 0;

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

  return result;
}
```

If length = 0, the upper bound in the loop condition becomes -1, but since length is unsigned, the whole expression is interpreted as an unsigned number. When -1 is interpreted as an unsigned number it becomes UMAX. Any i value is always <= UMAX, so this loop will never stop

Case Study - Ariane 5

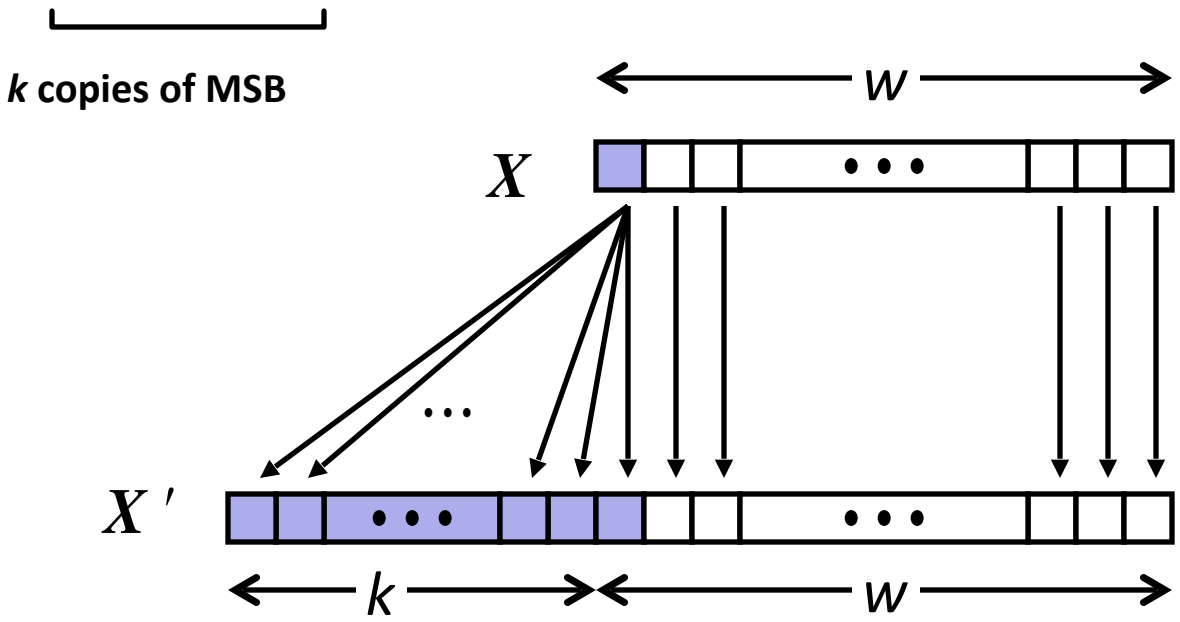
Youtube video: https://bit.ly/3bQJzma

Today: Bits, Bytes, and Integers

- **Integers**
  - Representation: unsigned and signed
  - Conversion, casting
  - **Expanding**

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



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