Today's Topics

- **Representation of integers: unsigned and signed**
- **Casting**
- **Arithmetic and shifting**
- **Sign extension**

Encoding Integers

- **The hardware (and C) supports two flavors of integers:**
  - unsigned – only the non-negatives
  - signed – both negatives and non-negatives
- **There are only 2^W distinct bit patterns of W bits, so...**
  - Can't represent all the integers
  - Unsigned values are 0 ... 2^W-1
  - Signed values are -2^{W-1} ... 2^{W-1}-1

Unsigned Integers

- **Unsigned values are just what you expect**
  - $b_7b_6b_5b_4b_3b_2b_1b_0 = b_7 2^7 + b_6 2^6 + b_5 2^5 + \ldots + b_1 2^1 + b_0 2^0$
    - Interesting aside: $1+2+4+8+\ldots+2^{N-1} = 2^N - 1$

```
 00111111
+00000001
 01000000
```

```
 63
+ 1
 64
```

- **You add/subtract them using the normal "carry/borrow" rules, just in binary**

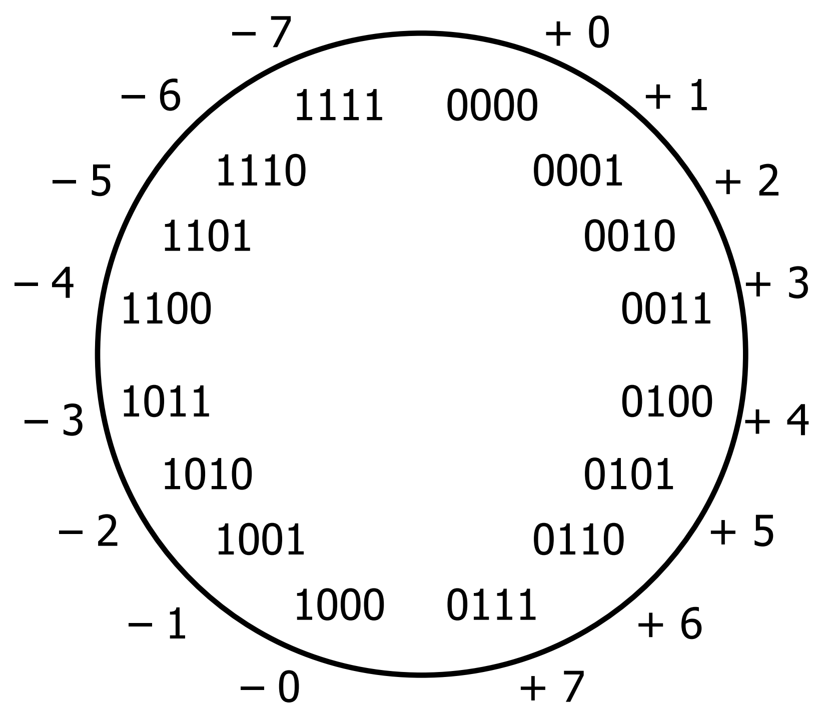
- **An important use of unsigned integers in C is pointers**
  - There are no negative memory addresses

Signed Integers

- **Let's do the natural thing for the positives**
  - They correspond to the unsigned integers of the same value
    - Example (8 bits): 0x00 = 0, 0x01 = 1, …, 0x7F = 127
- **But, we need to let about half of them be negative**
  - Use the high order bit to indicate 'negative'
  - Call it "the sign bit"
  - Examples (8 bits):
    - 0x00 = 00000000_2 is non-negative, because the sign bit is 0
    - 0x7F = 01111111_2 is non-negative
    - 0x80 = 10000000_2 is negative

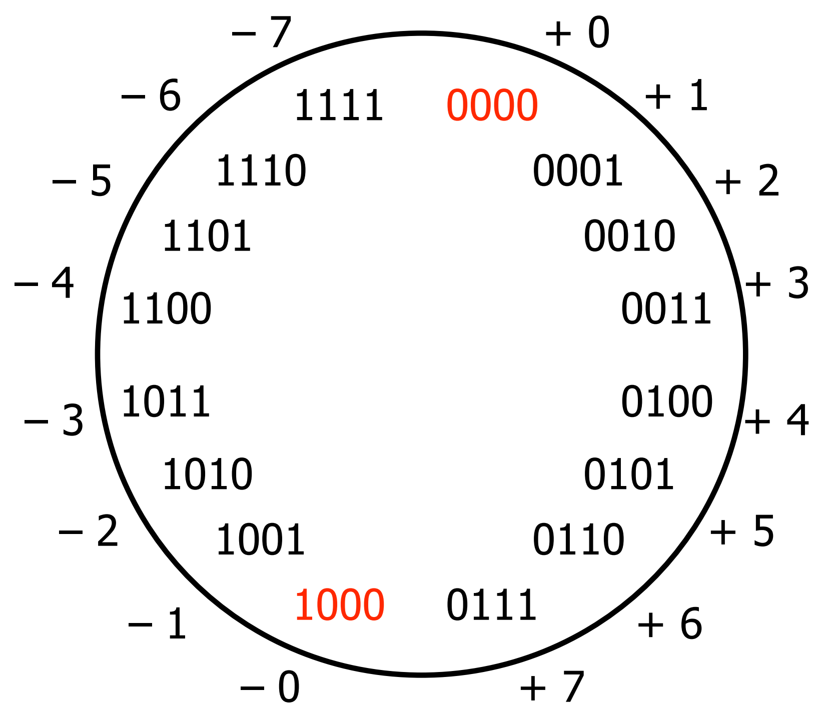
Sign-and-Magnitude Negatives

- **How should we represent -1 in binary?**
  - Possibility 1: 10000001_2
    Use the MSB for "+ or -", and the other bits to give magnitude



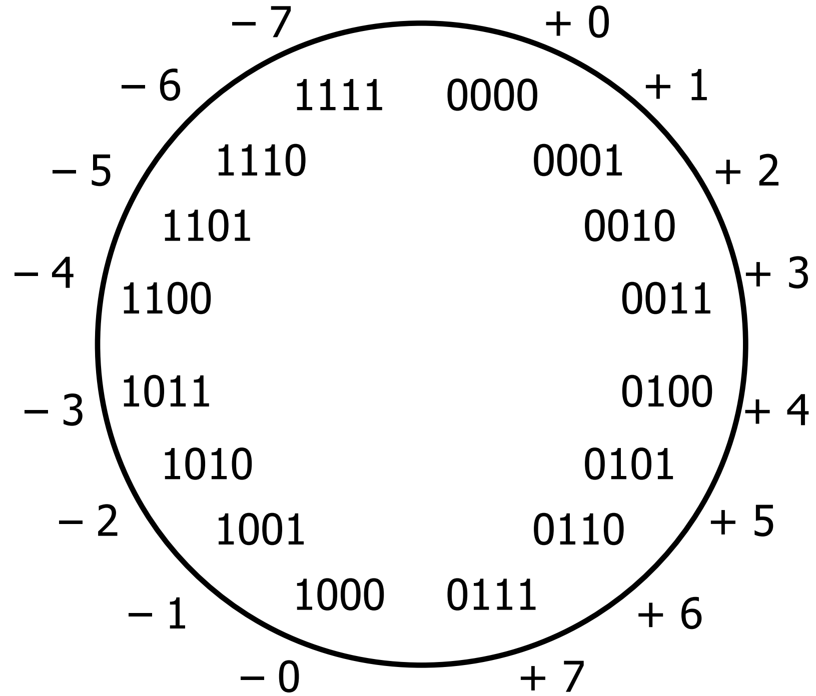
Sign-and-Magnitude Negatives

- **How should we represent -1 in binary?**
  - Possibility 1: 10000001_2
    Use the MSB for "+ or -", and the other bits to give magnitude
    (Unfortunate side effect: there are two representations of 0!)



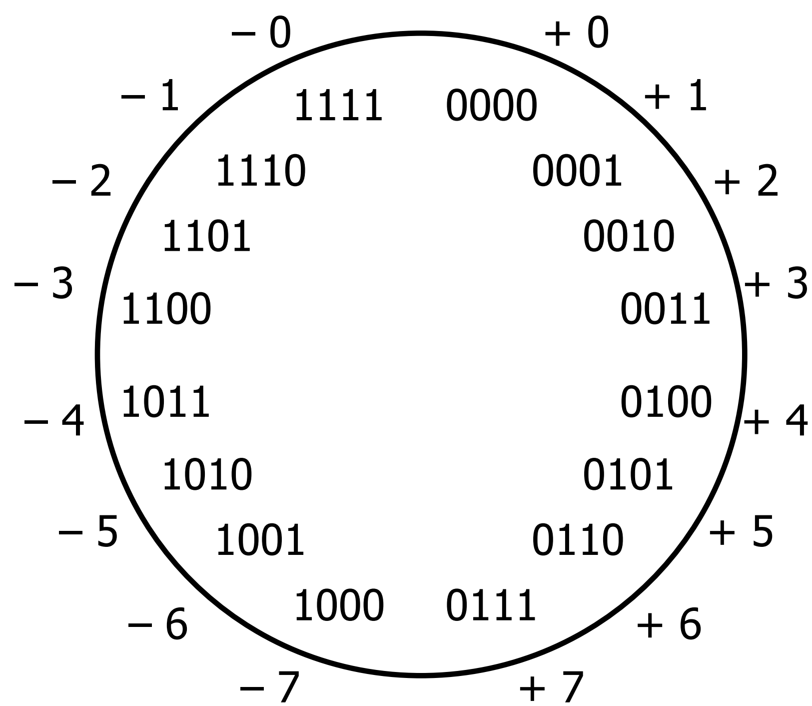
Sign-and-Magnitude Negatives

- **How should we represent -1 in binary?**
  - Possibility 1: 10000001_2
    Use the MSB for "+ or -", and the other bits to give magnitude
    Another problem: math is cumbersome
  - 4 – 3 != 4 + (-3)



Ones' Complement Negatives

- **How should we represent -1 in binary?**
  - Possibility 2: 11111110_2
    Negative numbers: bitwise complements of positive numbers
    It would be handy if we could use the same hardware adder to add signed integers as unsigned



Ones' Complement Negatives

- **How should we represent -1 in binary?**
  - Possibility 2: 11111110_2
    Negative numbers: bitwise complements of positive numbers
    - Solves the arithmetic problem

Add		Invert, add, add carry		Invert and add	
4	0100	4	0100	– 4	1011
+ 3	+ 0011	– 3	+ 1100	+ 3	+ 0011
= 7	= 0111	= 1	1 0000	– 1	1110
		add carry:	+1		
			= 0001		

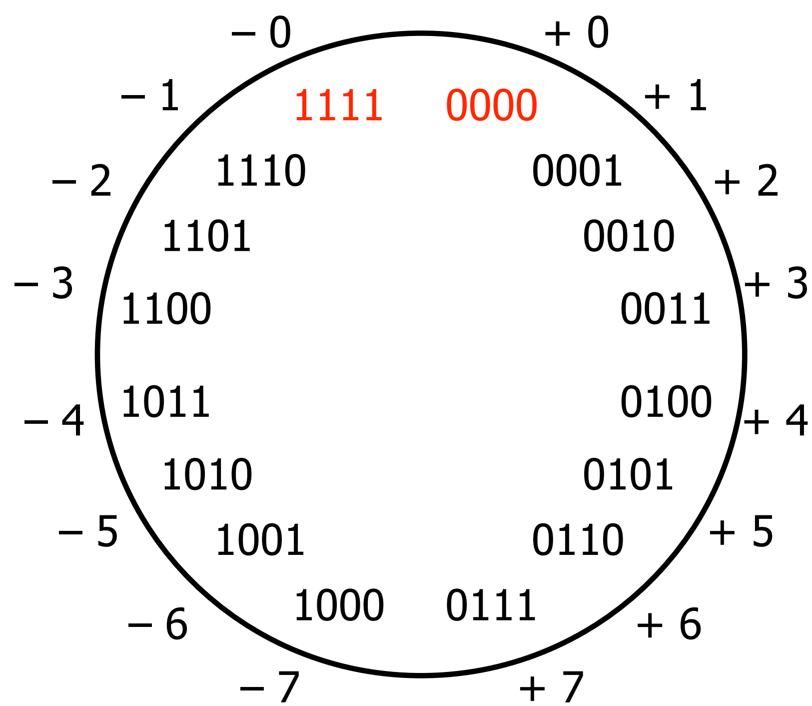
end-around carry

Ones' Complement Negatives

- **How should we represent -1 in binary?**
  - Possibility 2: 11111110_2
    Negative numbers: bitwise complements of positive numbers
    Use the same hardware adder to add signed integers as unsigned (but we have to keep track of the end-around carry bit)
  - Why does it work?
  - The ones' complement of a 4-bit positive number y is $1111_2 - y$
    - $0111 \equiv 7_{10}$
    - $1111_2 - 0111_2 = 1000_2 \equiv -7_{10}$
  - 1111_2 is 1 less than $10000_2 = 2^4 - 1$
    - $-y$ is represented by $(2^4 - 1) - y$

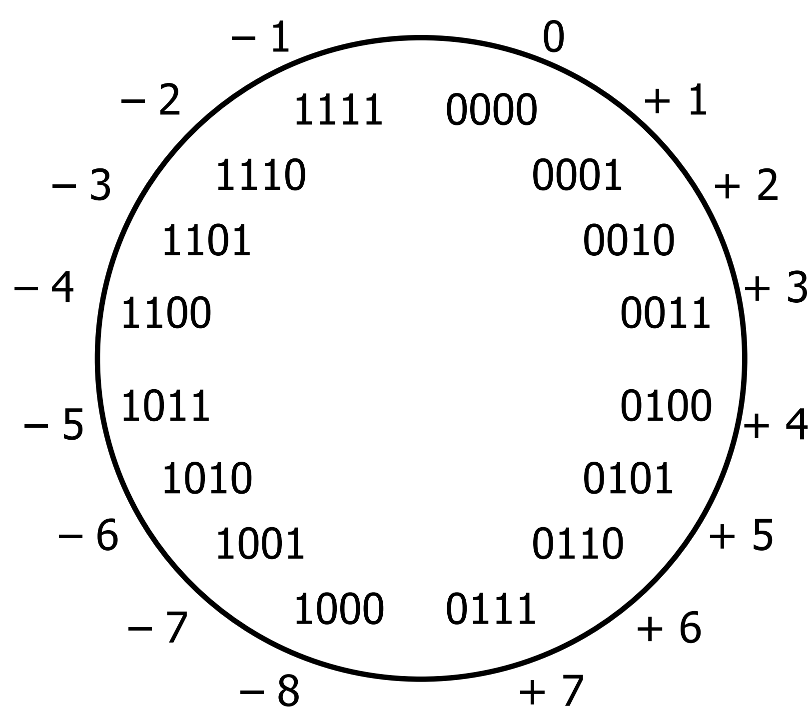
One's Complement Negatives

- **How should we represent -1 in binary?**
  - Possibility 2: 11111110_2
    Negative numbers: bitwise complements of positive numbers
    (But there are still two representations of 0!)



Two's Complement Negatives

- **How should we represent -1 in binary?**
  - Possibility 3: 11111111_2
    Bitwise complement plus one
    (Only one zero)



Two's Complement Negatives

- **How should we represent -1 in binary?**
  - Possibility 3: 11111111_2
    Bitwise complement plus one
    (Only one zero)
  - Simplifies arithmetic
    Use the same hardware adder to add signed integers as unsigned
    (simple addition; discard the highest carry bit)

Add		Invert and add		Invert and add	
4 + 3	0100 + 0011	4 – 3	0100 + 1101	– 4 + 3	1100 + 0011
= 7	= 0111	= 1 drop carry	1 0001 = 0001	– 1	1111

Two's Complement Negatives

- **How should we represent -1 in binary?**
  - Two's complement: Bitwise complement plus one

  - Why does it work?
  - Recall: The ones' complement of a b-bit positive number y is $(2^b – 1) – y$
  - Two's complement adds one to the bitwise complement, thus, -y is $2^b – y$
    - $–y$ and $2^b – y$ are equal mod 2^b
      (have the same remainder when divided by 2^b)
    - Ignoring carries is equivalent to doing arithmetic mod 2^b

Two's Complement Negatives

- **How should we represent -1 in binary?**
  - Two's complement: Bitwise complement plus one

    - What should the 8-bit representation of -1 be?

```
 00000001
+????????   (want whichever bit string gives right result)
 00000000

 00000010    00000011
+????????   +????????
 00000000    00000000
```

Unsigned & Signed Numeric Values

X	Unsigned	Signed
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

- Both signed and unsigned integers have limits
  - If you compute a number that is too big, you wrap: 6 + 4 = ? 15U + 2U = ?
  - If you compute a number that is too small, you wrap: -7 – 3 = ? 0U – 2U = ?
  - Answers are only correct mod 2^b
- The CPU may be capable of "throwing an exception" for overflow on signed values
  - It won't for unsigned
- But C and Java just cruise along silently when overflow occurs...

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	+16	11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

Numeric Ranges

- **Unsigned Values**
  - UMin = 0
    - 000…0
  - UMax = $2^w - 1$
    - 111…1
- **Two's Complement Values**
  - TMin = -2^{w-1}
    - 100…0
  - TMax = $2^{w-1} - 1$
    - 011…1
- **Other Values**
  - Minus 1
    - 111…1 0xFFFFFFFF (32 bits)

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

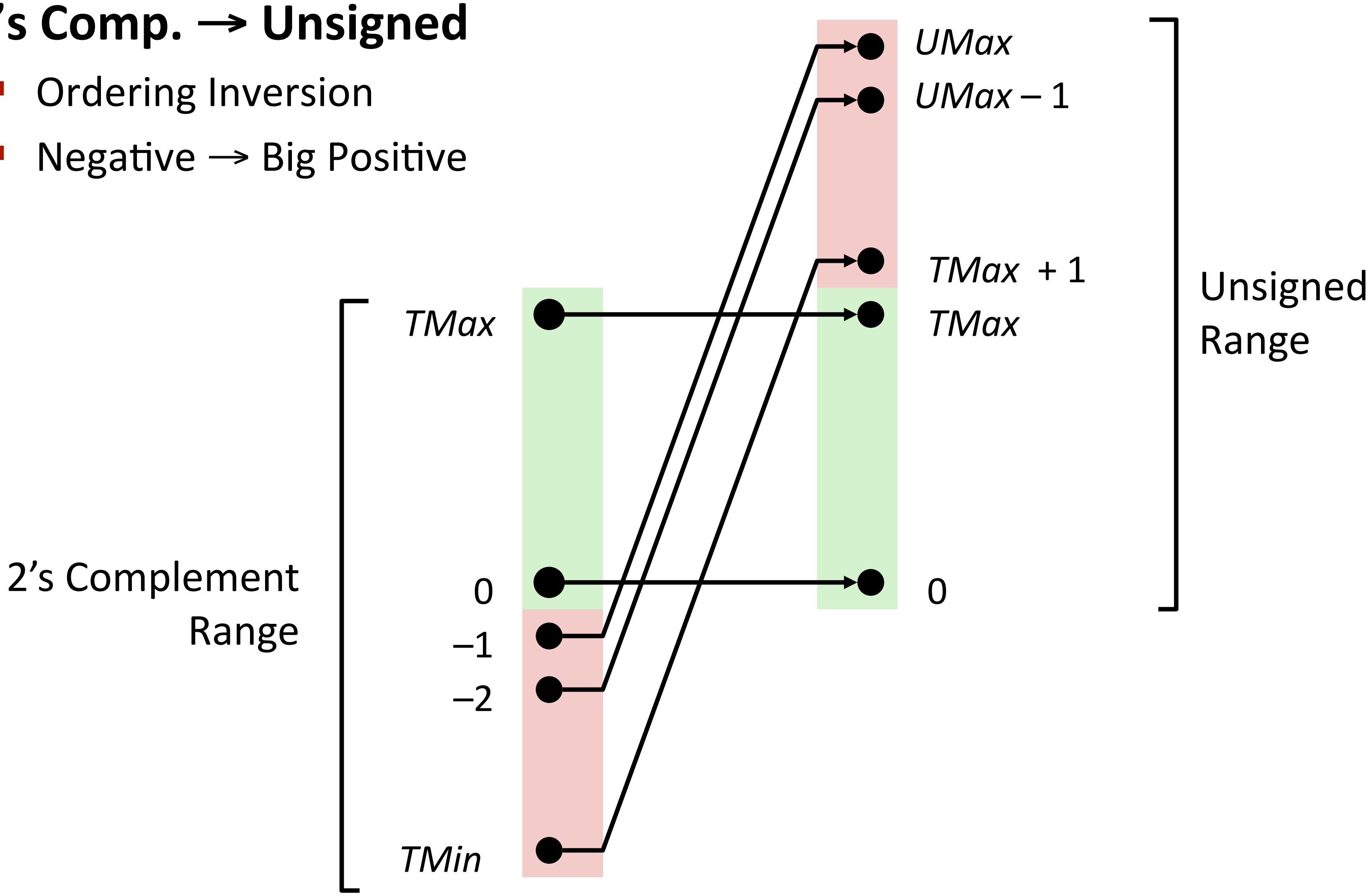
	W			
	8	**16**	**32**	**64**
UMax	255	65,535	4,294,967,295	18,446,744,073,709,551,615
TMax	127	32,767	2,147,483,647	9,223,372,036,854,775,807
TMin	-128	-32,768	-2,147,483,648	-9,223,372,036,854,775,808

- **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

Conversion Visualized

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



Signed vs. Unsigned in C

- **Constants**
  - By default are considered to be signed integers
  - Unsigned if have "U" as suffix
    - `0U`, `4294967259U`
- **Casting**
    - `int tx, ty;`
    - `unsigned ux, uy;`
  - Explicit casting between signed & unsigned same as U2T and T2U
    - `tx = (int) ux;`
    - `uy = (unsigned) ty;`
  - Implicit casting also occurs via assignments and procedure calls
    - `tx = ux;`
    - `uy = ty;`

21

Casting Surprises

- **Expression Evaluation**
  - If you mix unsigned and signed in a single expression, then ***signed values implicitly cast to unsigned***
  - Including comparison operations **<, >, ==, <=, >=**
  - Examples for *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

Shift Operations

- **Left shift: x << y**
  - Shift bit-vector x left by y positions
    - Throw away extra bits on left
    - Fill with 0s on right
  - Multiply by 2**y
- **Right shift: x >> y**
  - Shift bit-vector x right by y positions
    - Throw away extra bits on right
  - Logical shift (for unsigned)
    - Fill with 0s on left
  - Arithmetic shift (for signed)
    - Replicate most significant bit on left
    - Maintain sign of x
  - Divide by 2**y
  - Correct truncation (towards 0) requires some care with signed numbers

Argument x	01100010
<< 3	00010*000*
Logical >> 2	*00*011000
Arithmetic >> 2	*00*011000

Argument x	10100010
<< 3	00010*000*
Logical >> 2	*00*101000
Arithmetic >> 2	*11*101000

Undefined behavior when y < 0 or y ≥ word_size

Using Shifts and Masks

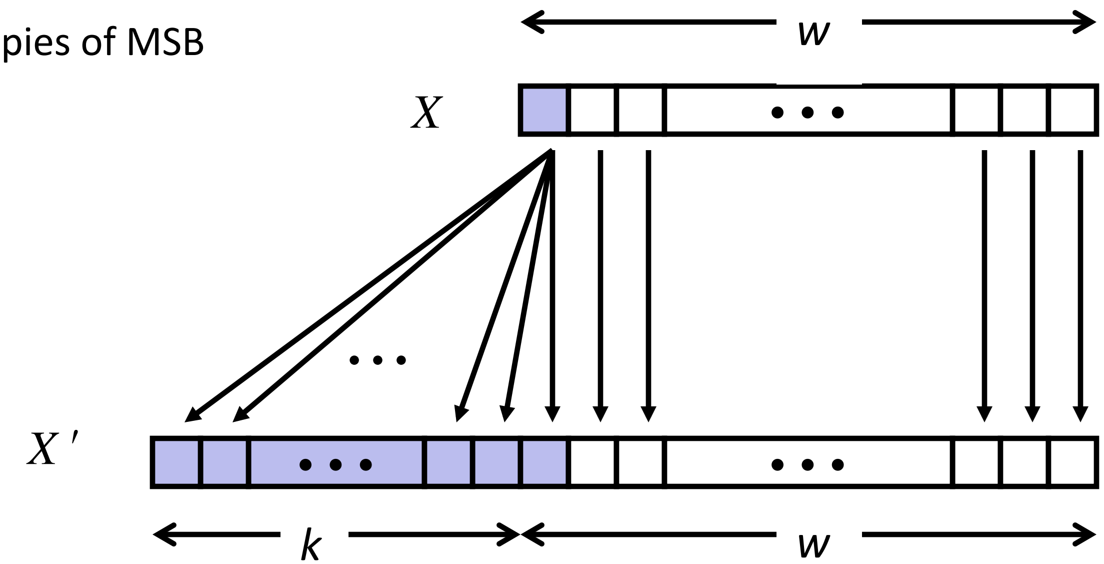
- **Extract 2nd most significant byte of an integer**
  - First shift: x >> (2 * 8)
  - Then mask: (x >> 16) & 0xFF

x	01100001 01100010 01100011 01100100
x >> 16	00000000 00000000 01100001 01100010
(x >> 16) & 0xFF	*00000000 00000000 00000000 11111111* 00000000 00000000 00000000 01100010

- **Extracting the sign bit**
  - (x >> 31) & 1 - need the "& 1" to clear out all other bits except LSB
- **Conditionals as Boolean expressions (assuming x is 0 or 1)**
  - if (x) a=y else a=z; which is the same as a = x ? y : z;
  - Can be re-written as: a = ((x << 31) >> 31) & y + (!x << 31) >> 31) & z;

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



Sign Extension Example

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

```
short int x =  12345;
int       ix = (int) x;
short int y = -12345;
int       iy = (int) y;
```

	Decimal	Hex	Binary
x	12345	30 39	00110000 01101101
ix	12345	00 00 30 39	00000000 00000000 00110000 01101101
y	-12345	CF C7	11001111 11000111
iy	-12345	FF FF CF C7	11111111 11111111 11001111 11000111