Integer Arithmetic and Undefined Behavior in C

Brad Karp
UCL Computer Science

CS 3007
23rd January 2018

(lecture notes derived from material from Eddie Kohler, John Regehr, Phil Gibbons, Randy Bryant, and Dave O'Hallaron)

Outline: Integer Arithmetic and Undefined Behavior in C

- C primitive data types
- C integer arithmetic
  - Unsigned and signed (two's complement) representations
  - Maximum and minimum values; conversions and casts
  - Perils of C integer arithmetic, unsigned and especially signed
- Undefined behavior (UB) in C
  - As defined in the C99 language standard
  - Consequences for program behavior
  - Consequences for compiler and optimizer behavior
  - Examples
  - Prevalence of UB in real-world Linux application-level code
- Recommendations for defensive programming in C

Example Data Representations

signed (default) and unsigned variants (applies to char, short, int, long)

C Data Type	Typical 32-bit	Typical 64-bit	x86-64
`char`	1	1	1
`short`	2	2	2
`int`	4	4	4
`long`	4	8	8
`float`	4	4	4
`double`	8	8	8
pointer	4	8	8

Portable C types with Fixed Sizes

C Data Type	all archs
{u}int8_t	1
{u}int16_t	2
{u}int32_t	4
{u}int64_t	8
uintptr_t	4 or 8

Type definitions available in **#include <stdint.h>**

Shift Operations

- Left Shift: `x << y`
  - Shift bit-vector **x** left **y** positions
    - – Throw away extra bits on left
    - Fill with 0's on right
- Right Shift: `x >> y`
  - Shift bit-vector **x** right **y** positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on left
- Undefined Behavior (on which more shortly…)
  - Shift amount < 0 or ≥ word size

Argument x	01100010
<< 3	
Log. >> 2	
Arith. >> 2	

Argument x	10100010
<< 3	
Log. >> 2	
Arith. >> 2	

Shift Operations

- Left Shift: `x << y`
  - Shift bit-vector **x** left **y** positions
      - – Throw away extra bits on left
    - Fill with 0's on right
- Right Shift: `x >> y`
  - Shift bit-vector **x** right **y** positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on left
- Undefined Behavior (on which more shortly…)
  - Shift amount < 0 or ≥ word size

Argument `x`	`01100010`
`<< 3`	`00010`
Log. `>> 2`	
Arith. `>> 2`	

Argument `x`	`10100010`
`<< 3`	
Log. `>> 2`	
Arith. `>> 2`	

Shift Operations

- Left Shift: `x << y`
  - Shift bit-vector **x** left **y** positions
      – Throw away extra bits on left
    - Fill with 0's on right
- Right Shift: `x >> y`
  - Shift bit-vector **x** right **y** positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on left
- Undefined Behavior (on which more shortly…)
  - Shift amount < 0 or ≥ word size

Argument x	01100010
<< 3	00010000
Log. >> 2	
Arith. >> 2	

Argument x	10100010
<< 3	
Log. >> 2	
Arith. >> 2	

Shift Operations

- Left Shift: `x << y`
  - Shift bit-vector **x** left **y** positions
    - – Throw away extra bits on left
    - Fill with 0's on right
- Right Shift: `x >> y`
  - Shift bit-vector **x** right **y** positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on left
- Undefined Behavior (on which more shortly…)
  - Shift amount < 0 or ≥ word size

Argument x	01100010
<< 3	00010000
Log. >> 2	011000
Arith. >> 2	

Argument x	10100010
<< 3	
Log. >> 2	
Arith. >> 2	

Shift Operations

- Left Shift: `x << y`
  - Shift bit-vector **x** left **y** positions
    - – Throw away extra bits on left
    - Fill with 0's on right
- Right Shift: `x >> y`
  - Shift bit-vector **x** right **y** positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on left
- Undefined Behavior (on which more shortly…)
  - Shift amount < 0 or ≥ word size

Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	

Argument x	10100010
<< 3	
Log. >> 2	
Arith. >> 2	

Shift Operations

- Left Shift: `x << y`
  - Shift bit-vector **x** left **y** positions
    - – Throw away extra bits on left
    - Fill with 0's on right
- Right Shift: `x >> y`
  - Shift bit-vector **x** right **y** positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on left
- Undefined Behavior (on which more shortly…)
  - Shift amount < 0 or ≥ word size

Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	011000

Argument x	10100010
<< 3	
Log. >> 2	
Arith. >> 2	

Shift Operations

- Left Shift: `x << y`
  - Shift bit-vector **x** left **y** positions
    - – Throw away extra bits on left
    - Fill with 0's on right
- Right Shift: `x >> y`
  - Shift bit-vector **x** right **y** positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on left
- Undefined Behavior (on which more shortly…)
  - Shift amount < 0 or ≥ word size

Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	00011000

Argument x	10100010
<< 3	
Log. >> 2	
Arith. >> 2	

Shift Operations

- Left Shift: `x << y`
  - Shift bit-vector **x** left **y** positions
    - – Throw away extra bits on left
    - Fill with 0's on right
- Right Shift: `x >> y`
  - Shift bit-vector **x** right **y** positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on left
- Undefined Behavior (on which more shortly…)
  - Shift amount < 0 or ≥ word size

Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	00011000

Argument x	10100010
<< 3	00010
Log. >> 2	
Arith. >> 2	

Shift Operations

- Left Shift: `x << y`
  - Shift bit-vector **x** left **y** positions
    - – Throw away extra bits on left
    - Fill with 0's on right
- Right Shift: `x >> y`
  - Shift bit-vector **x** right **y** positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on left
- Undefined Behavior (on which more shortly…)
  - Shift amount < 0 or ≥ word size

Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	00011000

Argument x	10100010
<< 3	00010000
Log. >> 2	
Arith. >> 2	

Shift Operations

- Left Shift: `x << y`
  - Shift bit-vector **x** left **y** positions
    - – Throw away extra bits on left
    - Fill with 0's on right
- Right Shift: `x >> y`
  - Shift bit-vector **x** right **y** positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on left
- Undefined Behavior (on which more shortly…)
  - Shift amount < 0 or ≥ word size

Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	00011000

Argument x	10100010
<< 3	00010000
Log. >> 2	101000
Arith. >> 2	

Shift Operations

- Left Shift: `x << y`
  - Shift bit-vector **x** left **y** positions
    - – Throw away extra bits on left
    - Fill with 0's on right
- Right Shift: `x >> y`
  - Shift bit-vector **x** right **y** positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on left
- Undefined Behavior (on which more shortly…)
  - Shift amount < 0 or ≥ word size

Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	00011000

Argument x	10100010
<< 3	00010000
Log. >> 2	00101000
Arith. >> 2	

Shift Operations

- Left Shift: `x << y`
  - Shift bit-vector **x** left **y** positions
      - Throw away extra bits on left
    - Fill with 0's on right
- Right Shift: `x >> y`
  - Shift bit-vector **x** right **y** positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on left
- Undefined Behavior (on which more shortly…)
  - Shift amount < 0 or ≥ word size

Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	00011000

Argument x	10100010
<< 3	00010000
Log. >> 2	00101000
Arith. >> 2	101000

Shift Operations

- Left Shift: `x << y`
  - Shift bit-vector **x** left **y** positions
    - – Throw away extra bits on left
    - Fill with 0's on right
- Right Shift: `x >> y`
  - Shift bit-vector **x** right **y** positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on left
- Undefined Behavior (on which more shortly…)
  - Shift amount < 0 or ≥ word size

Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	00011000

Argument x	10100010
<< 3	00010000
Log. >> 2	00101000
Arith. >> 2	11101000

Integer Numeric Ranges

Integer Numeric Ranges

- Unsigned Values
  - *UMin* = 0

    000…0
  - *UMax* = $2^w - 1$

    111…1

Integer 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
  - negative 1
    111…1

Integer 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
  - negative 1
    111…1

Values for word size 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

Integer Ranges 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`**, etc.
  - Values platform specific
  - Also, in **`<stdint.h>`** **`INT{8,16,32,64}_{MIN,MAX}`** and **`UINT{8,16,32,64}_MAX`**

Unsigned & Signed Integer 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
- ⇒ Can Invert Mappings
  - $U2B(x) = B2U^{-1}(x)$
    - Bit pattern for unsigned integer
  - $T2B(x) = B2T^{-1}(x)$
    - Bit pattern for two's comp integer

Mapping Signed ↔ Unsigned (W=4)

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 (W=4)

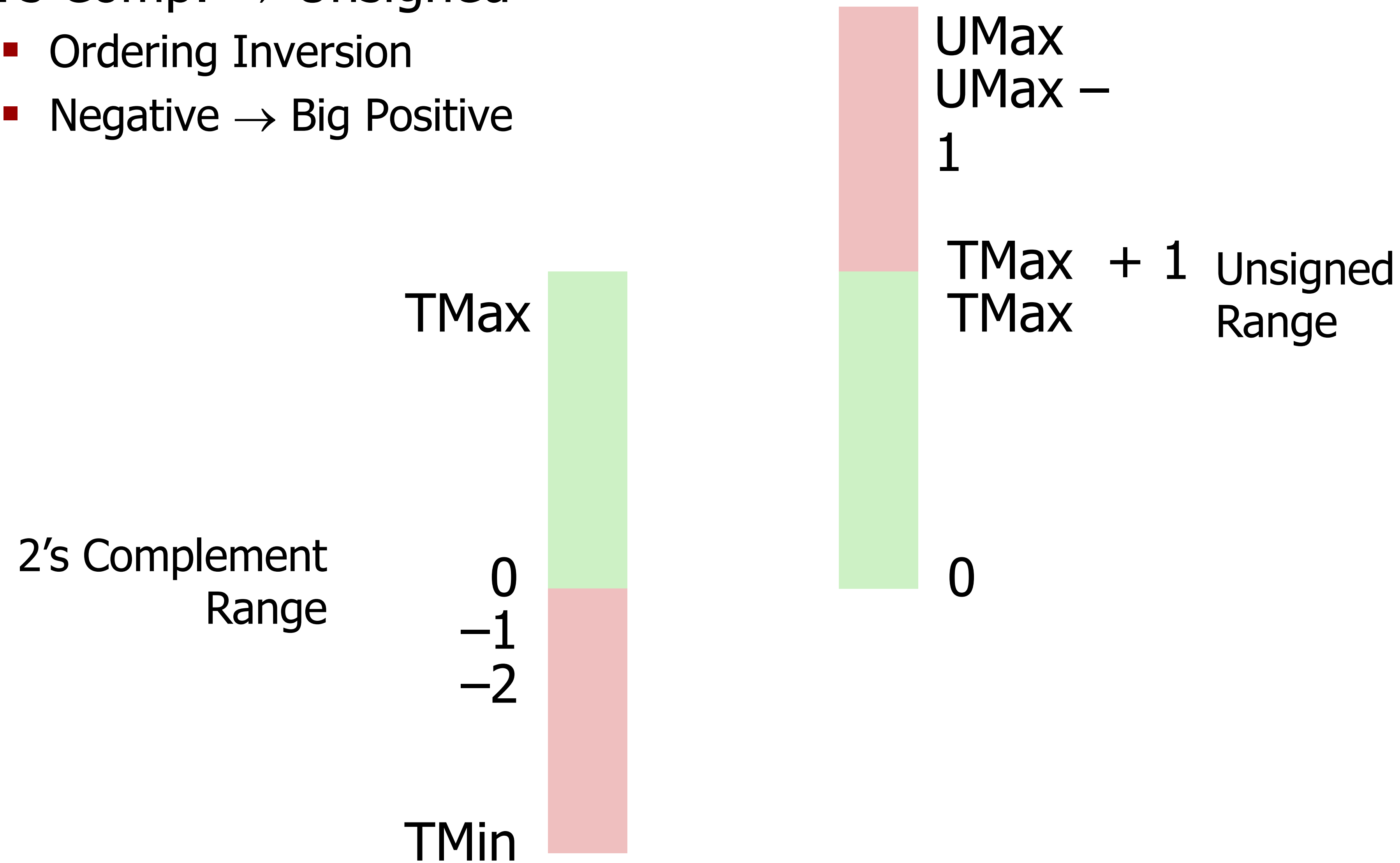
Bits	Signed	Unsigned
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	-8	8
1001	-7	9
1010	-6	10
1011	-5	11
1100	-4	12
1101	-3	13
1110	-2	14
1111	-1	15

Mapping Signed ↔ Unsigned (W=4)

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

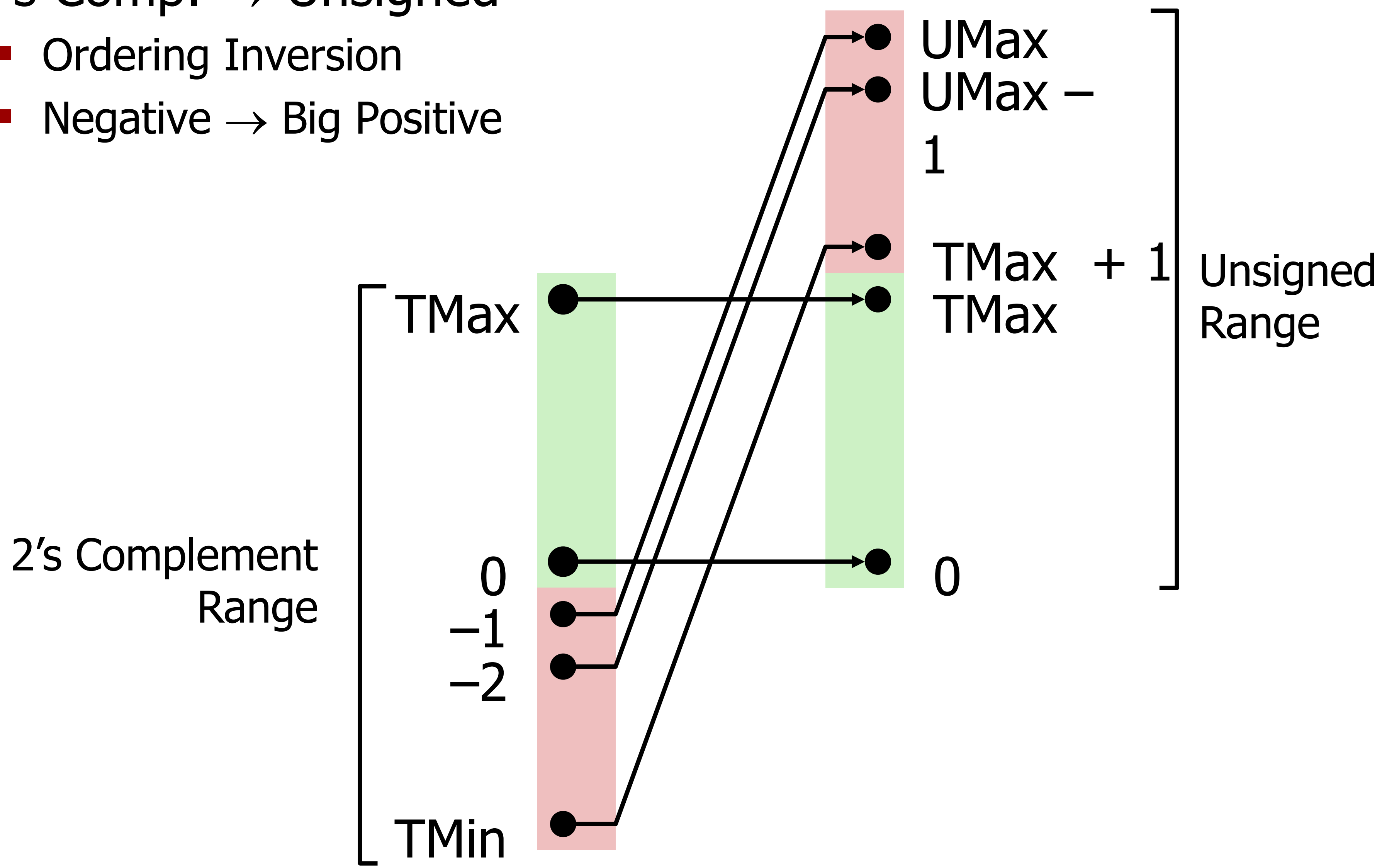
Conversion Visualized

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



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
  - Explicit casting between signed & unsigned just **takes bits as-is and reinterprets their value using the other representation**

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

  - Implicit casting also occurs via assignments and procedure calls

```
tx = ux;                    int fun(unsigned u);
uy = ty;                    uy = fun(tx);
```

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 for $W = 32$:
    TMIN = -2,147,483,648 ,   TMAX = 2,147,483,647

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

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 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

Unsigned vs. Signed: Easy to Make Mistakes

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

Unsigned vs. Signed: Easy to Make Mistakes

```
unsigned 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)
  . . .
```

Safely Using an **`unsigned`** Loop Index

- Broken:

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

Safely Using an `unsigned` Loop Index

- Broken:

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

- Safe:

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

Safely Using an `unsigned` Loop Index

- Broken:

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

- Safe:

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

- Why is the latter safe?
  - because `0U - 1 == UINT_MAX` in unsigned arithmetic!

Summary: Casting Signed ↔️ Unsigned

- Bit pattern is maintained
- But reinterpreted
- Can have unexpected effects: adding or subtracting 2^w

- Beware expressions mixing **`signed`** and **`unsigned int`**
  - `int` implicitly cast to `unsigned`!!

Stepping Back: Undefined Behavior in C

- Many operations in C offer defined behavior, where (e.g.,) the C99 standard specifies exactly what result the C abstract machine will produce
- Some (alas, fairly many) operations in C are explicitly noted by the C99 standard as undefined behavior
- In the words of the C99 standard (ISO be praised):

Stepping Back: Undefined Behavior in C

- Many operations in C offer defined behavior, where (e.g.,) the C99 standard specifies exactly what result the C abstract machine will produce
- Some (alas, fairly many) operations in C are explicitly noted by the C99 standard as undefined behavior
- In the words of the C99 standard (ISO be praised):

undefined behavior: behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

NOTE Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

EXAMPLE An example of undefined behavior is the behavior on integer overflow.

Stepping Back: Undefined Behavior in C

- Many operations in C offer defined behavior, where (e.g.,) the C99 standard specifies exactly what result the C abstract machine will produce
- Some (alas, fairly many) operations in C are explicitly noted by the C99 standard as undefined behavior
- In the words of the C99 standard (ISO be praised):

undefined behavior: behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements

NOTE Possible undefined behavior ranges from ignoring the situation completely with unpredictable results, to behaving during translation or program execution in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message), to terminating a translation or execution (with the issuance of a diagnostic message).

EXAMPLE An example of undefined behavior is the behavior on integer overflow.

- **199 undefined behaviors** in C99!

Broad Categories of UB in C (most common, but not exhaustive!)

- Programmer error involving pointers and memory allocation/deallocation
  - walking past the end of an array; use-after-free(); double free(); memory leaks; &c.
- Integer overflow
  - in unsigned arithmetic, exceeding UINT_MAX is defined; wrap to zero
  - similarly for subtracting 0U – 1U; wrap to UINT_MAX
  - in signed arithmetic, though, overflow is undefined behavior
  - Why?
    - Historically, different CPU architectures represented signed integers differently, and yielded different results upon overflow in signed integer arithmetic.
  - So why is *unsigned* overflow defined?
    - Because unsigned representations and overflow results didn't/don't vary across CPU architectures!

Wait. Why *deliberately* design a language to include UB???

- Performance
  - possible for compiler to generate instructions that explicitly detect some UB cases and generate run-time errors, and render such behavior defined
  - e.g., check shift less than word length; check for overflow on every signed arithmetic operation; bounds checks on array accesses; &c.
  - cost:
    - "30-50% performance reduction" on tight loops [Regehr, others]
    - optimized Rust 15-20% slower than optimized C for loops that do signed integer arithmetic
- Simplifies compiler design and implementation
  - Compiler needn't reason about results generated by complex "corner cases"; enjoin programmer from writing such code

And what are the drawbacks of a language with 199 UBs?

- More difficult to write correct programs
  - Programmers often unaware of UBs, unintentionally write code that exercises them
  - Can think of such code as "buggy"
  - …but the compiler can behave in surprising ways when it encounters code identifiable as exercising UB
  - …because the C99 spec says that anything can happen once a program exercises UB (silent incorrect results; expected results; termination; etc.), C compilers assume programmers don't write code that exercises UB
- Trend in recent years: compilers may **discard code** when they detect UB
  - Again, C99 says that incorrect results are fine once UB invoked…so discarding code may be totally consistent with C99
  - Less code means faster programs; perverse incentive

Integer Undefined Behavior in C

- Unsigned arithmetic generally defined; one exception is x / 0 (also UB for signed)
- Signed arithmetic UB:
  - Overflow for addition, subtraction, negation
    - compiler assumes machine result is valid (it may not be)
  - Overflow for multiplication
  - Overflow for integer division and remainder (mod)
    - Consider INT_MIN / -1
    - Remember, one more negative value in two's complement than there are positive values!
- Remember: the compiler may assume that no program ever produces any of the above results!

Example: The Case of the Dropped Assertion

```
int check_signed_increment(int x) {
  assert(x + 1 > x);
  return x + 1;
}

int main(int argc, char** argv) {
  int x = strtol(argv[1], NULL, 0);
  int x_incr = checked_signed_increment(x);
  printf("%d + 1 = %d\n", x, x_incr);
}
```

- Compile without optimization, invoke with 0x7fffffff:
  - result is crash (assertion failure), as overflow from 0x7fffffff to 0x80000000 (INT_MIN) when computing **`x + 1`**
- Compile with optimization, invoke with 0x7fffffff:
  - Result is output "**`2147483647 + 1 = -2147483648`**"!
  - Assertion code dropped by compiler! Why: **`x + 1 > x`** always true, if signed overflow cannot be executed by programmer!

Example: Programmer Errs; Compiler Takes Creative License

```
struct tun_struct *tun = ...;
struct sock *sk = tun->sk;
if (!tun)
  return POLLERR;
/* write to address based on tun */
```

- Real code from Linux kernel (CVE-2009-1897)
- Programmer dereferences tun before null pointer check
- Compiler notes dereference in second line, concludes "**`tun != NULL`**; if it were, programmer would have implemented undefined behavior"
- Compiler does not emit any code for if statement!
- Result: exploit that allows attacker to elevate privilege in Linux (because of later use of **`NULL tun`**)

Example: UB Throwdown, Postgres Devs vs. MIT PhD Student

```
int64_t arg1 = ...;              // user input
int64_t arg2 = ...;              // user input
if (arg2 == 0)                   // prevent division by zero
  ereport(ERROR, ...);
int64_t result = arg1 / arg2;   // division of signed 64-bit ints
if (arg2 == -1 && arg1 < 0 && result <= 0)  // UB!
  ereport(ERROR, ...);
```

- Context: check for signed division overflow in POSTgres SQL database; division precedes check
- Programmer error: incorrect expectation that integer division overflow of -2^{63} / -1 in C "wraps" from 2^{63} (unrepresentable in signed 64-bit integer) to -2^{63}
  - check `result <= 0` intended to catch this
  - Java behaves that way: defines signed integer division overflow to "wrap"
  - but in C, all signed overflow is UB

Example: UB Throwdown, Postgres Devs vs. MIT PhD Student

```
int64_t arg1 = ...;              // user input
int64_t arg2 = ...;              // user input
if (arg2 == 0)                   // prevent division by zero
  ereport(ERROR, ...);
int64_t result = arg1 / arg2;   // division of signed 64-bit ints
if (arg2 == -1 && arg1 < 0 && result <= 0)  // ALWAYS FALSE (!)
  ereport(ERROR, ...);
```

- Compiler notes computation of **`arg1 / arg2`**, both signed quantities…
- …and concludes expression in final **`if`** must always be false: when **`arg2 == -1`** and **`arg1 < 0`**, **`result <= 0`** implies the prior division was UB, which "cannot be"
- x86-64 **`idivq`** instruction traps upon overflow
- Consequence: user can issue SQL query to database that provokes integer divide overflow, crashing database

Example: UB Throwdown, Postgres Devs vs. MIT PhD Student

```
int64_t arg1 = ...;              // user input
int64_t arg2 = ...;              // user input
if (arg2 == 0)                   // prevent division by zero
  ereport(ERROR, ...);
int64_t result = arg1 / arg2;   // division of signed 64-bit ints
if (arg2 == -1 && arg1 < 0 && result <= 0)  // ALWAYS FALSE (!)
  ereport(ERROR, ...);
```

- MIT PhD student working on tools to detect UB in C finds above bug, submits patch to POSTgres team:

```
int64_t arg1 = ...;              // user input
int64_t arg2 = ...;              // user input
if (arg2 == 0)                   // prevent division by zero
  ereport(ERROR, ...);
int64_t result;
if (arg1 == INT64_MIN && arg2 == -1) // check for overflow first
  ereport(ERROR, ...);                // report error on UB
else
  result = arg1 / arg2;               // only divide if safe
```

Example: UB Throwdown, Postgres Devs vs. MIT PhD Student

- Postgres team thanks reporter for report, but rejects proposed fix in favor of its own design:

```
int64_t arg1 = ...;                    // user input
int64_t arg2 = ...;                    // user input
int64_t result;
if (arg1 != 0 && (-arg1 < 0) == (arg1 < 0)) // @!$%!!
  ereport(ERROR, ...);                 // report error on UB
else
  result = arg1 / arg2;                 // only divide if safe
```

- Alas, Postgres team's fix still invokes UB!
  - **(-arg1 < 0)** attempts to detect whether **arg1** is -2^{63}, but does so by (as in the original broken code) assuming that overflow when negating this value "wraps" to a negative value…
  - …when computing 2^{63} in signed arithmetic is UB, so compiler concludes that **(-arg1 < 0)** is always false

UB That Causes "Unstable" Code: It's Out There

UB condition	# reports	# packages
null pointer dereference	59,230	2,800
buffer overflow	5,795	1,064
signed integer overflow	4,364	780
pointer overflow	3,680	614
oversized shift	594	193
aliasing	330	70
overlapping memory copy	227	47
division by zero	226	95
use after free	156	79
other libc (cttz, ctlz)	132	7
absolute value overflow	86	23
use after realloc	22	10

- Universe of Debian wheezy packages
- [Wang, Zeldovich et al., SOSP 2013]

More on Undefined Behavior in C

- John Regehr's (Utah) humbling "Quiz About Integers in C":
  - https://web.archive.org/web/20120604210447/https://blog.regehr.org/archives/721
- Regehr's 3-part series of blog posts on UB from 2010:
  - https://blog.regehr.org/archives/213
  - https://blog.regehr.org/archives/226
  - https://blog.regehr.org/archives/232
- Regehr et al. on the incidence of integer overflow in real, widely used C and C++ code:
  - http://www.cs.utah.edu/~regehr/papers/tosem15.pdf
- Regehr and Cuoq's blog post on the state of UB in 2017, including great sanitizer tools to detect UB in your code:
  - https://blog.regehr.org/archives/1520
- Wang, Zeldovich et al. (MIT) on overzealous C optimizers and UB, and a tool to detect silently dropped code (2013):
  - https://pdos.csail.mit.edu/papers/stack:sosp13.pdf