Sewanee: The University of the South

CS 370: Computer Organization

Integer Representations and Operations

Instructor:
Professor Stephen P. Carl

Sewanee: The University of the South

How do we represent Negative Numbers?

So far, unsigned numbers

Obvious solution: define leftmost bit to be sign

- 0 ⇒ +, 1 ⇒ -
- Rest of bits can be numerical value of number

This representation is called sign and magnitude

For 32-bit integers. $+1_{10}$ would be:

0000 0000 0000 0000 0000 0000 0000 0001

And -1_{10} in sign and magnitude would be:

1000 0000 0000 0000 0000 0000 0000 0001

Sewanee: The University of the South

Shortcomings of sign and magnitude

Arithmetic circuit is complicated

- Special steps required to determine whether numbers have the same sign (heeeeey, what's your sign?)

Also, two zeros

- 0x00000000 = $+0_{ten}$
- 0x80000000 = -0_{ten}
- What would two 0s mean for programming?

Sign and magnitude representation was abandoned (for the most part)

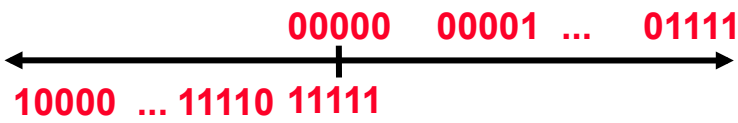
Sewanee: The University of the South

Another try: complement the bits

Example: $7_{10} = 00111_2$ $-7_{10} = 11000_2$

This representation is called One's Complement

Note: positive numbers have leading 0s, negative numbers have leadings 1s.



What is 11111?

How many positive numbers in N bits?

How many negative ones?

Sewanee: The University of the South

Shortcomings of One's complement

Arithmetic still is somewhat complicated.

Still two zeros

- `0x00000000` = $+0_{ten}$
- `0xFFFFFFFF` = -0_{ten}

Although used for awhile on some computer systems, one's complement was eventually abandoned because another solution was better….

Sewanee: The University of the South

Another Attempt …

Thought Experiment: Car Odometer

- 99998 ➔ 99999 ➔ 00000 ➔ 00001 ➔ 00002 ➔ 00003
- Idea of "rolling over to 0"

Binary Odometer:

- 11110 ➔ 11111 ➔ 00000 ➔ 00001 ➔ 00010 ➔ 00011
- In decimal: -2 ➔ -1 ➔ 0 ➔ 1 ➔ 2 ➔ 3

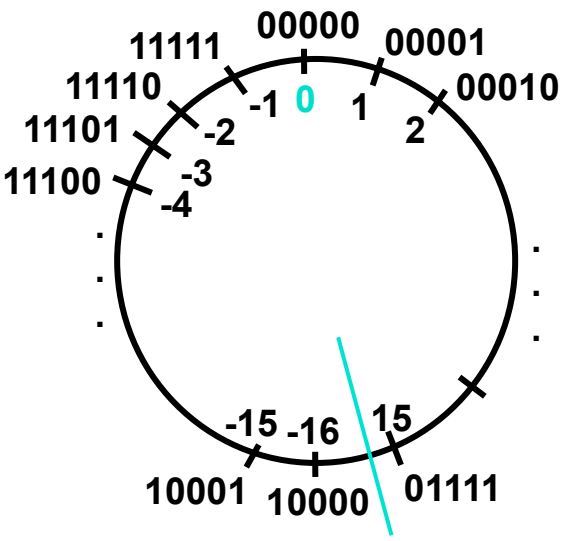
With no obvious better alternative, pick representation that makes the math simple:

- $11111_2 == -1_{10}$ $11110_2 == -2_{10}$ $11101_2 == -3_{10}$

This representation is called Two's Complement

Sewanee: The University of the South

2's Complement Number "line" for 5 bit numbers



- 2^{N-1} non-negatives (0..15 here)
- 2^{N-1} negatives
- one zero
- how many positives?

Sewanee: The University of the South

Two's Complement for N=32

$0000 \ldots 0000\ 0000\ 0000\ 0000_{two}$ =	0_{10}
$0000 \ldots 0000\ 0000\ 0000\ 0001_{two}$ =	1_{10}
$0000 \ldots 0000\ 0000\ 0000\ 0010_{two}$ =	2_{10}
. . .	
$0111 \ldots 1111\ 1111\ 1111\ 1101_{two}$ =	$2{,}147{,}483{,}645_{10}$
$0111 \ldots 1111\ 1111\ 1111\ 1110_{two}$ =	$2{,}147{,}483{,}646_{10}$
$0111 \ldots 1111\ 1111\ 1111\ 1111_{two}$ =	$2{,}147{,}483{,}647_{10}$
$1000 \ldots 0000\ 0000\ 0000\ 0000_{two}$ =	$-2{,}147{,}483{,}648_{10}$
$1000 \ldots 0000\ 0000\ 0000\ 0001_{two}$ =	$-2{,}147{,}483{,}647_{10}$
$1000 \ldots 0000\ 0000\ 0000\ 0010_{two}$ =	$-2{,}147{,}483{,}646_{10}$
. . .	
$1111 \ldots 1111\ 1111\ 1111\ 1101_{two}$ =	-3_{10}
$1111 \ldots 1111\ 1111\ 1111\ 1110_{two}$ =	-2_{10}
$1111 \ldots 1111\ 1111\ 1111\ 1111_{two}$ =	-1_{10}

- Most significant bit called the sign bit
- 1 "extra" negative; no positive $2{,}147{,}483{,}648_{ten}$

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

Encoding Example (Cont.)

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

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**

Sewanee: The University of the South

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

- Negative 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

Sewanee: The University of the South

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

Sewanee: The University of the South

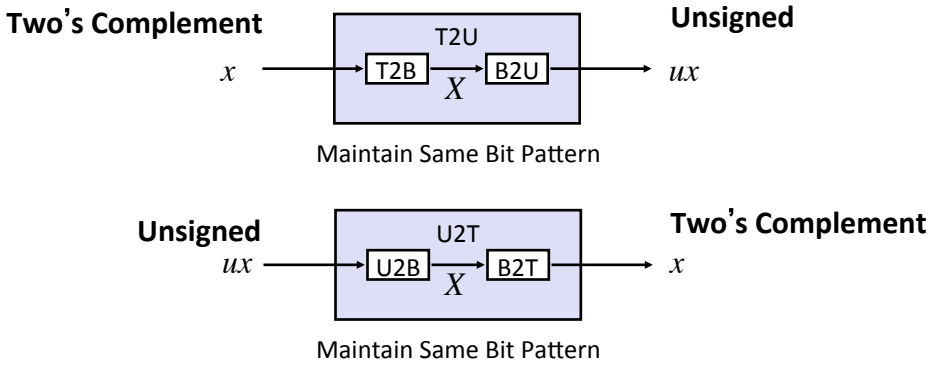
Unsigned & Signed Numeric Values

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**
  - Nonnegative encodings are the same
- **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

Sewanee: The University of the South

Mapping Between Signed & Unsigned



- **Mappings been unsigned and two's complement numbers:**
  keep bit representations and reinterpret

Sewanee: The University of the South

Mapping Signed ↔ Unsigned

Bits	Signed		Unsigned
0000	0		0
0001	1		1
0010	2		2
0011	3		3
0100	4		4
0101	5	T2U →	5
0110	6		6
0111	7	← U2T	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

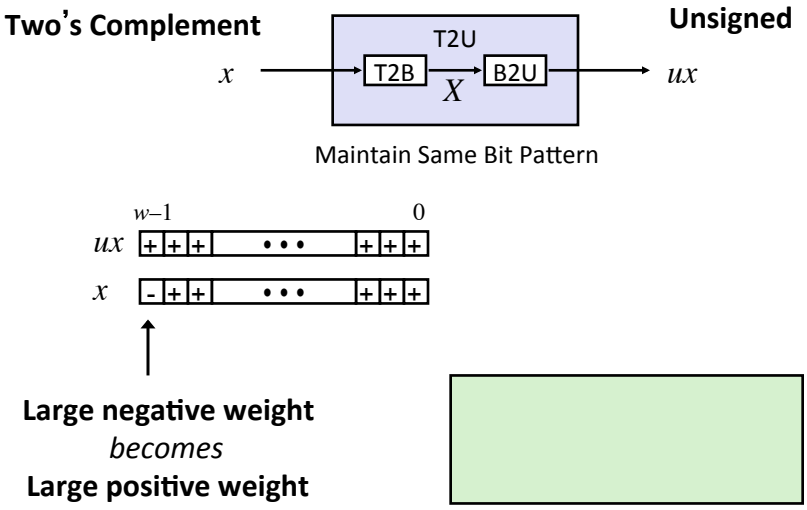
Sewanee: The University of the South

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

Sewanee: The University of the South

Relation between Signed & Unsigned

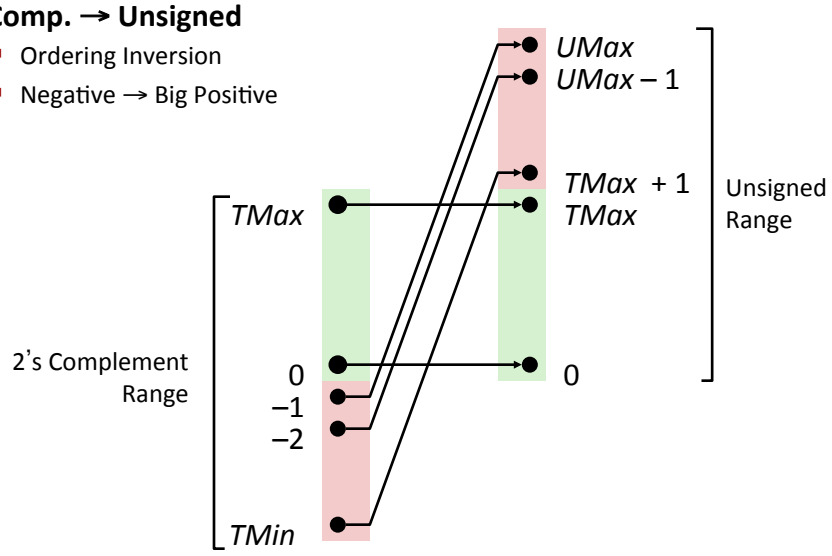


Sewanee: The University of the South

Conversion Visualized

2's Comp. → Unsigned

- Ordering Inversion
- Negative → Big Positive



Signed vs. Unsigned in C

- **Literals**
  - All constants considered to be signed integers by default
  - Force to unsigned using "U" as suffix

```
0U, 4294967259U
```

- **Casting**
  - Explicit casting between 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 Surprise!

In Arithmetic and Logic Expressions:

- When mixing unsigned and signed in single expression, ***signed values are implicitly cast to unsigned***
- This includes all arithmetic and comparison operations: **<, >, ==, <=, >=**
- Examples for *W* = 32: **TMIN = -2,147,483,648 , TMAX = 2,147,483,647**

Code Security Example

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

The trick here is the function signature of `memcpy`:

```
/* Declaration of library function memcpy */
void *memcpy(void *dest, void *src, size_t n);
```

`size_t` is the *unsigned* integer type which is returned by the `sizeof` operator

Malicious Usage

```
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```
#define MSIZE 528

void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    . . .
}
```

`maxlen` can be negative, but when `len` is passed to `memcpy`, it is cast to unsigned

Summary: Casting Signed ↔ Unsigned

- Bit pattern is maintained
- But reinterpreted
- Can have the unexpected effect of adding/subtracting 2^w

- In expressions containing signed and unsigned int
  `int` is cast to `unsigned`!!

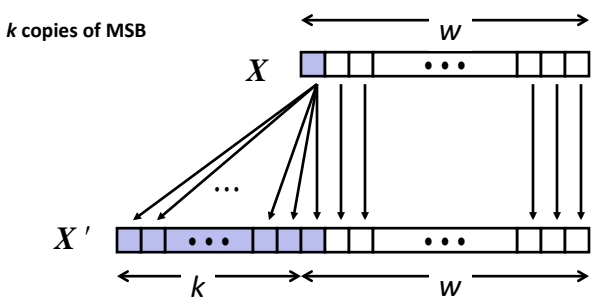
Expanding: Sign Extension

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

When converting from a smaller to larger integer data type, C automatically performs sign extension

Expanding, Truncating: Basic Rules

- **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 behaviour

Negation: Complement & Increment

Claim: the following Holds for 2's Complement Integers:

```
~x + 1 == -x   <- look closely!
```

Observation: `~x + x == 1111…111 == -1`

	x	1 0 0 1 1 1 0 1
+	~x	0 1 1 0 0 0 1 0
	-1	1 1 1 1 1 1 1 1

Complement & Increment Examples

x = 15213

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
~x	-15214	C4 92	11000100 10010010
~x+1	-15213	C4 93	11000100 10010011
y	-15213	C4 93	11000100 10010011

x = 0

	Decimal	Hex	Binary
0	0	00 00	00000000 00000000
~0	-1	FF FF	11111111 11111111
~0+1	0	00 00	00000000 00000000