CS 4400
Computer Systems

LECTURE 3

Representing integers

Integer arithmetic

Encoding Integers

- Two different ways bits can be used to encode integers:
  - *unsigned* – only nonnegative numbers represented
  - *signed* – negative, zero, and positive values represented

- Both encodings represent a finite range of integers.

type declaration (C)	min	max
`char`	-128	127
`unsigned char`	`0`	255
`short`	-32768	32767
`unsigned short`	`0`	65535
`int`	-2147483648	2147483647
`unsigned int`	`0`	4294967295

Unsigned Integers

- Let vector $\vec{x} = [x_{w-1}, x_{w-2},\ldots, x_0]$ denote a *w*-bit integer value.
- Treat $\vec{x}$ as a number written in binary notation to obtain the unsigned interpretation. $B2U_w(\vec{x}) = \sum_{i=0}^{w-1} x_i 2^i$
- $UMin_w = [00\ldots00] = 0$
- $UMax_w = [11\ldots11] = 2^w-1$
- $B2U_w: \{0,1\}^w \rightarrow \{0, \ldots, 2^w-1\}$
- *Bijection*—associates a unique value to each *w*-bit vector.

Signed Integers

- The most common computer representation of signed integers is *two's complement*.

$$B2T_w(\vec{x}) = -x_{w-1}\,2^{w-1} + \sum_{i=0}^{w-2} x_i\,2^i$$

- *Sign bit*—the MSB, 1: negative and 0: nonnegative.
- $TMin_w = [10...00] = -2^{w-1}$
- $TMax_w = [01...11] = 2^{w-1} - 1$
- $B2T_w$: $\{0,1\}^w \rightarrow \{-2^{w-1}, ..., 2^{w-1} - 1\}$
- Is $B2T_w$ a bijection?

Exercises: Encoding Integers

Let $w = 4$.

Hex	Binary	$B2U_w$	$B2T_w$
A	[1010]	$2^3 + 2^1 = 10$	$-2^3 + 2^1 = -6$
B			
C			
D			
E			
F			

Clicker Question

If you have ResponseCard clicker, channel is **41**.

If you are using ResponseWare, session id is **CS1400U**.

Let $w = 8$. Compute $B2T_w$ for hex AE.

A. -84

B. -82

C. -62

D. 84

E. 174

F. none of the above

More on Two's Complement

- The two's complement range is asymmetric.
- $\mathrm{UMax}_w > 2 * \mathrm{TMax}_w$. Why?
- Both encodings represent numeric value 0 the same way.
- The C standard does not require two's complement for signed integers.
  - Nearly all machines use it anyway. Does this affect portability?
  - See `limits.h` for constants delimiting ranges of different integer data types for a particular compiler and machine.
  - Other ways of representing signed integers?

Clicker Question

Which of the following expressions is equivalent to `~x` ?

A. `x`

B. `-x`

C. `x + 1`

D. `-x + 1`

E. `-x – 1`

F. none of the above

Signed-Unsigned Conversions

- Since both $B2U_w$ and $B2T_w$ are bijections, they have well-defined inverses, $U2B_w$ and $T2B_w$.
- Consider $U2T_w(\vec{x}) = B2T_w(U2B_w(\vec{x}))$.
  - Takes number between 0 and 2^{w-1}, yields number between -2^{w-1} and $2^{w-1}-1$.
  - Both numbers have identical bit representations.
- Conversely, consider $T2U_w(\vec{x}) = B2U_w(T2B_w(\vec{x}))$.
- *Example* (8-bit): [10101010], 170 unsigned, -86 signed
- How do these functions affect signed and unsigned in C?

Unsigned and Signed in C

```
int x = -1;
unsigned ux = (unsigned) x;  // ux is UMax_w
```

- In C, values are signed unless
  - indicated with type (e.g., `unsigned short`)
  - `U` in constant (e.g., `1234U`)
  - doesn't fit in `signed long`
  - starts `0x` and doesn't fit in `signed int`
- Use conversion codes `%d` (or `%i`), `%u` to print signed and unsigned decimal values, respectively.
- When signed and unsigned values are mixed in expressions, the signed values are promoted to unsigned.

Example: Unsigned and Signed

unsigned_signed.c

```
#include <stdio.h>

int main(void) {
  int tx, ty;
  unsigned ux, uy;

  tx = -96;
  uy = 15;

  ux = (unsigned) tx;  // explicit cast to unsigned
  ty = uy;             // implicit cast to signed

  printf("%d, %d, %u, %u\n", tx, ty, ux, uy);

  printf("%d\n", ux + ty);  // WHY = -81??

  return 0;
}
```

```
unix> gcc unsigned_signed.c
unix> ./a.out
-96, 15, 4294967200, 15
-81
```

Clicker Question

Assume `x` and `y` are arbitrary `int` values.

```
(x > 0) || (-x >= 0)
```

A. always true

B. sometimes true

C. always false

D. I don't know

Expanding Bit Representations

- A common operation is to convert between integers of different word sizes, retaining the same numeric value.
- To convert from smaller word size to larger:
  - for unsigned, simply add leading 0s – *zero extension*
  - for signed, add leading Xs such that X=MSB – *sign extension*
- *Example:*

```
short sx = 12345;   // 0x3039
short sy = -12345;  // 0xCFC7
int x = sx;         // 0x00003039
int y = sy;         // 0xFFFFCFC7
```

Truncating Bit Representations

- To convert from larger word size to smaller (*w*-bit to *k*-bit, where $w > k$):
  - drop high-order w-k bits – *truncation*
- Truncation of a number can alter its value, a form of overflow.

```
short x = (int) 12345;  // 0x00003039, x is 12345
short y = (int) 53191;  // 0x0000CFC7, y is -12345
```

- For unsigned x, truncation to k-bit equivalent to x mod 2^k. For signed x?

Advice on Unsigned

- Implicit casts are tricky (because they are easy to overlook) and can lead to bugs.
- To avoid such bugs, one might consider using only signed values.
  - Few languages other than C support unsigned values.
  - Java supports only signed values, requires two's complement, and guarantees that `>>` is an arithmetic shift.
- Unsigned values are very useful when thought of as a collection of bits (flags), with no math interpretation.

Unsigned Addition

- Consider w-bit unsigned values x and y, $0 \leq x, y \leq 2^w$-1.
  - Representing the sum could require w+1 bits, $0 \leq x + y \leq 2^{w+1}$-2
- In math, we cannot place any bound on the word size required to fully represent the results of arithmetic ops.
- Unsigned arithmetic is a form of modulo arithmetic.
  - Unsigned addition is equivalent to $(x + y) \bmod 2^w$.
  - `unsigned_add(`x, y`)` $= x + y$, if $x + y < 2^w$
  - `unsigned_add(`x, y`)` $= x + y - 2^w$, if $2^w \leq x + y < 2^{w+1}$
- *Example:*

```
unsigned short x = 65530 + 6;  // x is 0
```

Overflow

- An arithmetic operation is said to *overflow* when the full integer result cannot fit within the limits of the data type.
- In C, overflow is not signaled as an error.
  - Some types of overflow <u>may</u> be signaled with a warning.
- We know that overflow has occurred during unsigned integer addition $s = x + y$, if $s < x$ (equivalently, if $s < y$).
- *Example:*

```
unsigned x = ~0;
unsigned y = 2;
unsigned s = x + y;
if(s < x) { ... }    // overflow
```

Two's Complement Addition

- Consider w-bit values x and y, $-2^{w-1} \leq x, y \leq 2^{w-1}-1$.
  - Representing the sum could require w+1 bits, $-2^{w} \leq x + y \leq 2^{w}-2$
- We must truncate the result to w bits.
  - However, this is not as familiar as modulo arithmetic.
- The w-bit sum is the same as for unsigned addition.

$$\mathrm{U2T}_w(\ [(x + y) \bmod 2^w]\ )$$

- Both positive and negative overflow can occur.
- *Example*:

```
int x = 1 << 30;
int y = x + x;     // y is -2147483648
y = -++x -x;       // y is 2147483646
```

Cases of Overflow

- *Negative overflow*—if $-2^w \le x + y < -2^{w-1}$.
  - both x and y must be negative
  - a nonnegative integer is the result (counter to usual math)
  - `twoscomp_add(`x, y`)` $= x + y + 2^w$
- *No overflow*—if $-2^{w-1} \le x + y < 2^{w-1}$.
  - `twoscomp_add(`x, y`)` $= x + y$
- *Positive overflow*—if $2^{w-1} \le x + y < 2^w$.
  - both x and y must be positive
  - a negative integer is the result (counter to usual math)
  - `twoscomp_add(`x, y`)` $= x + y - 2^w$

Unsigned Multiplication

- Consider w-bit unsigned values x and y, $0 \le x, y \le 2^w-1$.
  - The product could require $2w$ bits, $0 \le x * y \le (2^w-1)^2$
- We must truncate the result to w bits.
  - In C, the low-order w bits are retained as the result.
  - Equivalent to computing the product mod 2^w.
- *Example*:

```
unsigned short x = 1 << 15;  // 32768
x *= 3;                      // 32768
```

Two's Complement Multiplication

- Consider w-bit values x and y, $-2^{w-1} \le x, y \le 2^{w-1}-1$.
  - The product could require $2w$ bits, $-2^{2w-2} + 2^{w-1} \le x * y \le 2^{2w-2}$
- We must truncate the result to w bits.
  - However, this is not as familiar as for unsigned multiplication.
- The w-bit product is the same as for unsigned multiply.

$$\mathrm{U2T}_w(\ [(x * y) \bmod 2^w]\ )$$

- *Example*:

```
short x = 1 << 14;              // 16384
unsigned short y = x * 3;       // 49152
x *= 3;                         // -16384
```

Multiplication by Powers of Two

- Integer multiplication used to be slow ($\geq$ 12 cycles) compared to other integer operations.
  - addition, subtraction, bit-level ops, and shifts—1 cycle each
- An important (compiler) optimization was to replace multiplications by constant factors with shifts and adds.
- Let x be an integer. For any $k \geq 0$, $x * 2^k$ is equivalent to adding k 0's to the right of the bit representation of x.
- *Example*:

```
unsigned int = 11 << 3;     // 88
int = -11 << 3;             // -88
```

Division by Powers of Two

- Integer division was also slow ($\geq$ 30 cycles).
- Let x be an unsigned integer. For any $k \geq 0$, $x / 2^k$ is equivalent to adding k 0's to the left of the bit rep for x.
  - logical shift
- Let x be an signed integer. For any $k \geq 0$, $x / 2^k$ is equivalent to adding k b's to the left of the bit rep for x.
  - b is the value of x's MSB, arithmetic shift
  - What if $x < 0$?
- *Example:*

```
int x = 55 >> 3;      // 6
int y = -55 >> 3;     // -7 (should be -6)
```

Biasing

- If $x < 0$, integer division should round negative results up toward zero. Right shifting does not accomplish this.

- To correct for this improper rounding, we must "bias" the value before shifting.
  - First add 2^k-1 to x.

- For x, represented with two's complement and using arithmetic shifts, x / 2^k is equivalent to

```
(x<0 ? (x + (1<<k)-1) : x) >> k
```

- *Example*:

```
int y = (-55 + (1 << 3) – 1) >> 3;  // -6
```

Clicker Question

Assume `x` and `y` are arbitrary `int` values.

```
((y - x) << 3) - y - x == 7*y - 9*x
```

A. always true

B. sometimes true

C. always false

D. I don't know