Computer Systems
CS107

Cynthia Lee
Today’s Topics

LECTURE:

- Friday we did:
  - Integer representation
    - Unsigned numbers in base 2
    - Signed numbers with two’s complement
    - Hexadecimal

- Today:
  - Reasoning about special conditions with signed and unsigned
    - Overflow and underflow conditions
    - Comparison operators (< >) with signed and unsigned
    - Sign extension and truncation
  - Bytes, bits, bitwise operators
Review self-test: Two’s complement

What is the bit representation of the following variables?
(Reminder: “int” data type in C uses 32-bit two’s complement)

a) int x = -35;

b) int y = 22;
Review self-test: Two’s complement

What is the bit representation of the following variables?
(Reminder: “int” data type in C uses 32-bit two’s complement)

a) unsigned int a = 3500400123;

b) int b = 3500400123;
Reasoning about signed and unsigned
Integer Representation

(assume binary values shown are all 32 bits)
Signed and unsigned numbers

At which points can overflow occur for signed and unsigned int? (assume binary values shown are all 32 bits)

A. Signed and unsigned can both overflow at points X and Y
B. Signed can overflow only at X, unsigned only at Y
C. Signed can overflow only at Y, unsigned only at X
D. Signed can overflow at X and Y, unsigned only at X
E. Other
UNSIGNED integers

(assume binary values shown are all 32 bits)

≈+4 billion  0

Discontinuity means overflow possible here

More increasing positive numbers

Increasing positive numbers
SIGNED integers

Discontinuity means overflow possible here

Increasing positive numbers

Negative numbers becoming less negative (i.e., increasing)

≈+2billion

≈-2billion

(assume binary values shown are all 32 bits)
Comparison operators in signed and unsigned numbers

```
int s1, s2, s3;
unsigned int u1, u2, u3;
```

Are the following statements true?
(assume that variables are set to values that place them in the spots shown)

- `s3 > u3` Easy: true
- `s1 > s3` Easy: false
- `u1 > u3` Easy: true
- `s1 > u3` Hmmm!??!

C just needs to choose one or the other scheme to dominate. It chooses…drumroll…

`unsigned`

So this is **TRUE**.
HOME SELF-TEST:
Comparison operators in signed and unsigned numbers

```c
int s1, s2, s3, s4;
unsigned int u1, u2, u3, u4;
```

Which many of the following statements are true? (assume that variables are set to values that place them in the spots shown)

- s3 > u3
- u2 > u4
- s2 > s4
- s1 > s2
- u1 > u2
- s1 > u3
Type truncation in the char/short/int/long family

```c
int i1 = 0x8000007F; // = -2147483521
int i2 = 0x000000FF; // = 255
char s1 = i1; // = 0x7F = 127
char s2 = i2; // = 0xFF = -1
unsigned char u1 = i1; // = 0x7F = 127
unsigned char u2 = i2; // = 0xFF = 255
```

- Regardless of source or destination signed/unsigned type, truncation always just truncates
- This can cause the number to change drastically in sign and value
Type promotion in the char/short/int/long family

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>sc = 0xFF;</td>
<td>// 0xFF = -1</td>
</tr>
<tr>
<td>unsigned char</td>
<td>uc = 0xFF;</td>
<td>// 0xFF = 255</td>
</tr>
<tr>
<td>int</td>
<td>s1 = sc;</td>
<td>// 0xFFFFFFFF = -1</td>
</tr>
<tr>
<td>int</td>
<td>s2 = uc;</td>
<td>// 0x000000FF = 255</td>
</tr>
<tr>
<td>unsigned int</td>
<td>u1 = sc;</td>
<td>// 0xFFFFFFFF = 4,294,967,295</td>
</tr>
<tr>
<td>unsigned int</td>
<td>u2 = uc;</td>
<td>// 0x000000FF = 255</td>
</tr>
</tbody>
</table>

- Promotion always happens according to the source variable’s type
  - Signed: “sign extension” (copy MSB—0 or 1—to fill new space)
  - Unsigned: “zero fill” (copy 0’s to fill new space)

- Note: When doing <, >, <=, >= comparison between different size types, it will promote to the larger type
  - “int < char” comparison will implicitly (1) assign char to int according to these promotion rules, then (2) do “int < int” comparison
Bitwise operators and bits as individual booleans

MOVING BEYOND THE “INT” INTERPRETATION OF BITS
Bitwise operators

- You’ve seen these categories of operators in C/C++:
  - Arithmetic operators: +, -, *, /
  - Comparison operators: ==, !=, <, >, <=, >=
  - Logical operators: &&, ||, !
  - (C++ only) Stream insertion operators: <<, >>

- Now meet a new category:
  - Bitwise operators: &\, |\, ^\, \sim\, >>\, <<
## Bitwise operators

<table>
<thead>
<tr>
<th>char a = 0b</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0;</th>
</tr>
</thead>
<tbody>
<tr>
<td>char b = 0b</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0;</td>
</tr>
</tbody>
</table>

- **and, intersection**: \( a \& b \)
- **or, union**: \( a \mid b \)
- **xor, different?**: \( a \^ b \)
- **not**: \( \sim a \)
- **shift left**: \( a\ll 2 \)
- **shift right**: \( a\gg 3 \)
To be or ~(to be), that is the question

- Be careful! Many of our bitwise operators have logical operators that seem similar but are subtly different.
  - ! and ~ are both “not” operators—not the same!

- **Question: is this guaranteed to always print?**
  
  ```
  int i;
  scanf("%d", &i);
  if (!i == ~i) printf("same this time\n");
  ```

A. Yes, always prints
B. Sometimes prints, but not always
C. No, never prints
D. You lost me at the code version of Shakespeare