COMP 264 Fall 2015 Lecture Notes

R. I. Greenberg

September 14, 2015

1 Fixed Point Number (Integer) Representations

- Fixed point notation: same number of digits to the right of the decimal point (or "binary point" in base 2 or more generally "radix point").
- In computers, fixed-point notations usually for integers (with no digits to the right of the radix point), and that's what we will take up here.
- Two main notations:
  - Unsigned (non-negative) notation. Only non-negative numbers are represented. Regular base 2 representation.; An n-bit number can range from 0 (all bits 0) to $2^n - 1$ (all 1s).
  - Two's complement notation. Both positive and negative numbers can be represented. Two equivalent ways to describe:
    - Same place values as for non-negative notation, except that leftmost bit has negative place value. (Range for an n-bit number is from -2^{n-1} to $2^{n-1} - 1$.)
    - (a) Non-negative represented as in non-negative notation but must always begin with a 0. (b) To represent a negative value, start with a representation of its absolute value, complement every bit, and then add 1.
- (Another notation: sign-magnitude. Leftmost bit gives sign; rest give magnitude in non-negative notation. We'll revisit for floating point.)
- Same bit pattern *might* represent two different things in non-negative and two's complement notations.

Signed vs Unsigned in C, Extension, Truncation

- Constants: signed as before; for unsigned add a "u", e.g., `98765u`.
- Several ways conversions may occur.
  1. Explicit cast, e.g., `(unsigned) -1` same as `0xFFFFFFFFu`.
  2. `printf` conversion, e.g., `printf("%u",-1);`.
  3. Implicit through assignment, e.g., `unsigned u; u=-1;`
  4. Promotion in an expression. e.g., after `float f=5; int i=2;`, doing `f/i` gives floating point 2.5, whereas `5/2` is integer 2. Similarly, in an expression involving a signed and an unsigned, the signed is converted to unsigned; can be very unintuitive, e.g., `unsigned u=0u; if (-1>u) printf("yes");`.
- Converting to a larger data type simple for unsigned: pad with 0s. For signed in two's complement format, need *sign extension*: pad with msb.
- Order of conversions matters even if end up with same final data type.
- Cutting off leftmost bits of an unsigned number to leave k bits is same as computing the number mod 2^k; more complicated for signed.
- Many languages (e.g., Java) do not support unsigned numbers, since they can lead to some unintuitive results. But useful for packing words with Boolean flags, manipulating addresses, modular arithmetic, and multiprecision arithmetic.
- Can find range of different data types on your machine by looking at `limits.h` file. e.g., `INT_MAX`, `INT_MIN`, and `UINT_MAX`. Ditto with `INT=>SHRT` or `INT=>LONG`.

2 Integer Arithmetic

2.1 Addition and subtraction

Addition and subtraction are quite simple under either of the unsigned (non-negative) and two's-complement integer representations that we have seen.

2.1.1 Addition

- Straightforward grade school method but in binary of course.
- But must watch for *arithmetic overflow*.

- Overflow in non-negative notation: when add two numbers whose sum is larger than the maximum value ($2^n - 1$ for n-bit numbers). Easy to detect: when carry out of leftmost position is 1. Not a problem if what you want to do is compute mod 2^n.

- Overflow in two's complement notation: result that is too large *or* result that is too small (very negative). To detect: check whether carry out of most significant bit position equals carry in; overflow when these two carries are unequal. Note: overflow never occurs when adding a non-negative number to a non-positive number; only the sum of two positive numbers or two negative numbers has the possibility of producing overflow.

2.1.2 Subtraction

- Using either unsigned (non-negative) or two's complement notation, subtraction is easily converted to addition, i.e., $X - Y$ is $X + (-Y)$; just need to compute $-Y$.

- Can get $-Y$ by employing procedure used to represent a negative number in two's complement. That is to represent $-Y$ where Y is positive, we did the bitwise complement of Y's non-negative representation and then added 1. This procedure actually works to negate Y even if Y is a negative number already in two's complement notation (except for the overflow possibility of negating, e.g., -128 in 8 bits to get 128). Furthermore, this procedure works even if X and Y are in non-negative notation and we seek a result in non-negative notation, as long as overflow does not occur.

- Overflow from subtraction in non-negative notation when result less than 0. Detect by inspecting carry out of leftmost position as for addition, but now overflow when it's 0 instead of when it's 1.

- Overflow from subtraction in two's complement notation under same condition as for addition, when the carry in to the most significant bit does not equal carry out.

2.2 Multiplication and division

- In either unsigned (non-negative) or two's complement notation, product of two n-bit numbers could require $2n$ bits. C generally truncates to n bits, which yields same bit pattern whether multiplication is signed or unsigned. Unsigned results again good mod 2^n even if truncated; result with overflow less meaningful for signed.

- Multiplying by 2^k same as shifting left k bits.

- C does *integer division* on integers; i.e., round towards zero.

- For unsigned, dividing by 2^k same as logical right shift by k bits.

- To divide an integer in two's complement format by 2^k, need *arithmetic* right shift, but still off by one for a negative number. Correction: For a negative no., add $2^k - 1$ before shifting right k bits:

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