More on Computer Arithmetic

• What is $x \& -x$?
  
  - Let’s try a few values:
    
    \[
    \begin{align*}
    x = 0 & \rightarrow 0000_2 \& 0000_2 = 0000_2 = 0 \rightarrow \text{Maybe } x \& -x = 0 \ ? \\
    x = 1 & \rightarrow 0012_2 \& 1111_2 = 0001_2 = 1 \rightarrow \text{Maybe } x \& -x = x \ ? \\
    x = 2 & \rightarrow 0010_2 \& 1110_2 = 0010_2 = 2 \rightarrow \text{Looks good so far}... \\
    x = 5 & \rightarrow 0101_2 \& 1011_2 = 0011_2 = 3 \rightarrow \text{Nope}! \\
    x = 8 & \rightarrow 1000_2 \& 1000_2 = 1000_2 = 8 \rightarrow \text{Maybe } x \& -x = x \text{ when } x \text{ is even, but 1 when } x \text{ is odd?} \\
    x = 6 & \rightarrow 0110_2 \& 1010_2 = 0010_2 = 2 \rightarrow \text{Nope again!}
    \end{align*}
    \]
    
  - So what’s the real pattern?
    
    * Note that, except when $x = 0$, the result always has exactly one “on” bit.
    * Moreover, that bit is always the rightmost “on” bit in $x$.
    
    ∗ There’s our answer: $x \& -x$ returns the least significant 1 bit of $x$.
  
  - Proof: suppose that $x$ has the form $x = \underbrace{\alpha}_{\text{arbitrary string of 1s and 0s}} 100\ldots0$, where $\alpha$ is an arbitrary string of 1s and 0s, and the string of 0s at the end of $x$ has length $k \geq 0$.
    
    * The underlined 1 is $x$’s least significant 1 bit; there are no 1s to the right of it.
    * If we flip all the bits in $x$, we get: $\sim x = \underbrace{\sim \alpha}_{\text{is string of 1s}} 011\ldots1$
    * To get $\sim -x$ we calculate $\sim x + 1$; but since $\sim x$ ends in a string of 1s, adding 1 causes a ripple effect which flips all the bits up to the underlined bit:
      
      \[
      \sim -x = \sim x + 1 = (\underbrace{\sim \alpha}_{\text{is string of 1s}} 011\ldots1) + 1 = \underbrace{\sim \alpha}_{\text{is string of 1s}} 100\ldots0
      \]
      
      * The ripple carry ends at the underlined bit because $0 + 1 = 1$ with no carry, so the bits in $\sim \alpha$ are not affected.
    
    * Consider now what happens when we perform the operation $x \& -x$:
      
      - On the right end, both $x$ and $\sim x$ have a string of $k$ 0s; $00\ldots0 \& 00\ldots0 = 00\ldots0$
      - On the left end, $\overline{\alpha} \& \overline{\sim \alpha}$ must be 0 because the two strings have opposite bits.
      - At the underlined bit, both $x$ and $\sim x$ have a one; this bit, the least significant 1 bit of $x$, is the only bit in $x \& -x$ which evaluates to 1.
    
    - $x \& -x$ = the least significant 1 bit of $x$; if $x$ has $k$ 0s on the end, this value is $2^k$

• Multiplication & Division
  
  - $x \ll i = x \cdot 2^i$
    
    * Proof: suppose we have $x = \overline{\begin{array}{c}
    x_{w-1} \quad x_{w-2} \quad \ldots \quad x_1 \quad x_0
    \end{array}}$(\begin{array}{c}
    x \text{ is a string of bits } x_{w-1}x_{w-2}\ldots x_1x_0
    \end{array})$
      
      Then $y = x \ll i = \overline{\begin{array}{c}
    y_{w-1} \quad \ldots \quad y_{i+1} \quad y_i \quad y_{i-1} \quad \ldots \quad y_0
    \end{array}}$
      
      So, $y = \sum_{k=0}^{w-1} 2^k y_k = \sum_{k=i}^{w-1} 2^k x_{k-i} = \sum_{k=0}^{w-1-i} 2^i \cdot 2^k x_k = 2^i \sum_{k=0}^{2^k} x_k = 2^i x$
Similarly, \( x \gg i = \lfloor x / 2^i \rfloor = x / 2^i \) (in integer arithmetic, division always includes flooring)

* Most machines will calculate \( x \gg i \) much faster than \( x / 2^i \)

Finally, \( x \& (2^i - 1) = x \% 2^i \)

* Again, most machines will calculate the former expression more quickly than the latter
* Good compilers will change multiplications and divisions into \( \ll, \gg, \) and \& whenever possible

• Signed Arithmetic

  – Addition, subtraction, and multiplication each use the same bit patterns for signed arithmetic and unsigned arithmetic
    * For example: \( 1111_2 - 0001_2 = 1110_2 \), whether \( 1111_2 \) is being used to represent \(-1\) or \(15\).
  – Division does not use the same bit patterns for signed and unsigned arithmetic
    * \( 1111_2/0010_2 = 0111_2 \) in unsigned arithmetic \((15/2 = 7)\)
    * \( 1111_2/0010_2 = 0000_2 \) in signed arithmetic \((-1/2 = 0)\)
    * This is one of many reasons why division and modulus are the hardest (slowest) arithmetic operations for a computer
    * Many processors don’t even include a division operation; higher-level software uses simpler operations to simulate division

• Logical Operations

  – Many of the bitwise operations have analogous logical operations
  – Logical operations operate on truth values; they always evaluate to either 1 or 0
  – What is \(!!x\)?
    * It’s not \( x \) - you’re thinking of \( \sim \sim x = x \)
    * \(!!x = 0\) if \( x = 0 \), 1 otherwise; in other words, \(!!x\) is equivalent to \((x \neq 0)\)

Data Representation

• Big-Endian vs. Little-Endian

  – Given a value that requires two bytes - say 32767=0x7FFF, the largest signed short - and an address in memory \( A \), how should we store that value?
    * Turns out to be a bit of a religious war

  – **Big-Endian**: Store the most significant bits in \( A \) and lesser bits in later addresses

    \[
    \begin{array}{c|c|c|c}
    \text{0x00000000} & \text{A} & \text{A+1} & 2^{32} - 1 \\
    \hline
    \text{0x7F} & \text{0xFF} & & \\
    \end{array}
    \]

    * This is how data is arranged when it is being transmitted across the internet

  – **Little-Endian**: Store the least significant bits in \( A \) and greater bits in later addresses

    \[
    \begin{array}{c|c|c|c}
    \text{0x00000000} & \text{A} & \text{A+1} & 2^{32} - 1 \\
    \hline
    \text{0xFF} & \text{0x7F} & & \\
    \end{array}
    \]

    * This is how data is stored in most computers
• Arrays
  – Memory is like an enormous array of unsigned char
    * In C, arrays are represented as contiguously-allocated subsets of memory
  – Arrays are **homogeneous collections** of data; everything in the array is of the same type
  – Given an array \( x[] \) of type \( T \), where the address of the array is \( A \), the address of item \( i \) is:
    \[ \&x[i] = A + i \cdot \text{sizeof}(T) \]

• Structs
  – Structs are **heterogeneous collections** of data; they can store multiple different types
  – In C, structs are also stored as a contiguous block, but the
  – An example:
    ```c
    struct foo {
        int a; ← size 4 bytes
        char b; ← size 1 bytes
        unsigned char c; ← size 1 bytes
        int *p; ← size 4 bytes
    }
    ```
  – So the total size should be \( 4 + 1 + 1 + 4 = 10 \) bytes, right?
    * But if we print out the addresses of `some_foo.c` and `some_foo.p`, we see that there is a gap of 3 bytes instead of 1; there are two empty bytes inserted between them. These bytes are **padding**.
    * Padding exists to maintain **alignment**: the processor is better at loading values from addresses that are a multiple of that value’s type size.
      · So it is faster to load an int from an address that is a multiple of 4, and a short from an address that is a multiple of 2
    * The actual size of foo is \( 4 + 1 + 1 + 2 \) (so `some_foo.p` is properly aligned) + 4 = 12 bytes
  – What if we remove \( p \) from the definition of foo? No need to align so the size should be 6, right?
    * But the padding is still there! The size of foo is 8.
    * Why? The compiler includes the padding *just in case* we have multiple items of type foo stored in a row. Because the first element of foo has alignment 4, the padding should stay.
  – If we eliminate \( a \) as well, then the remaining size is 2. With only chars (alignment 1) in foo, there is no need for padding.

• Unions
  – Unions are **overlapping collections** of data
  – An example:
    ```c
    union foo { 
        int a;
        char b;
        unsigned char c;
        int *p;
    }
    ```
– Essentially a way to tell the compiler “I know that I’m using this data in multiple ways”
– The size and alignment of the union are the same as the size and alignment of the largest element
– All elements in the union have the same address

• Function Layout

– Consider the factorial function, which recursively calls itself:
  * Each call to the function creates a new, local version of the variable n
  * If we print out the address of n in each call to the function, we see that the address decreases each time
    ‧ The amount of decrease appears to be constant
  * What if we alter the function so that factorial(2) calls factorial(1) then, after the latter has returned, calls factorial(1) again?
    ‧ factorial(1)’s version of n is stored in the same address both times; in fact, the local variables of any function called by factorial(2) would be stored in that same address

– All of the above is a result of the way functions are laid out in memory:

<table>
<thead>
<tr>
<th>0x00000000</th>
<th>← ← ←</th>
<th>2^{32} – 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>global variables &amp; code</td>
<td>THE HEAP</td>
<td>main()'s local variables</td>
</tr>
</tbody>
</table>

* Local variables of functions are stored in the stack:
  ‧ Those variables belonging to main() are stored at some high value in memory.
  ‧ Those variables belonging to functions called by main() are stored in slightly lower addresses; those belonging to those functions’ called functions in still lower addresses; etc...
* Global variables and the code itself are stored at very low values in memory
* Everything in between is the heap