Winter 2021 CS 33 Lecture Notes

Kyle Chui

2021-3-27

Contents

1 Bits and Bytes

1.1 Representing Data in Bits and Bytes

1.1.1 Everything is Bits

- Every bit is either a 0 or a 1.
- We can use sets of bits to not only tell the computer what to do (give it instructions), but also represent data.
- We use bits because they are easy to store, and reliably transmitted on noisy/inaccurate wires.

Example. *Counting in Binary*
We can use bits to represent numbers in base 2, or the binary number system. Thus we can use a set of bits to encode any number we want.

1.1.2 Encoding Byte Values

Definition. *Byte*
A *byte* is 8 bits, and can be thought of as a string of 0's and 1's of length eight.

- In binary, a byte can range from 00000000_2 to 11111111_2.
- In decimal, a byte can range from 0_{10} to 255_{10}.
- In hexadecimal (base 16), a byte can range from 00_{16} to FF_{16}.

Note. In hexadecimal, once you finish using the digits 1 through 9, you use the letters A through F to represent values of 10_{10} through 15_{10}. In C, we append "0x" before a string to indicate that it is a hexadecimal number (which may either be upper-case or lower-case). For example, $FA1D37B_{16}$ would be written as "0xFA1D37B" or "0xfa1d37b".

All data is just a long string of bits, so any value that we get out of it depends on the *context* of what we are reading in (a `double`, `char`, `int`, etc).

1.2 Bit Manipulation

1.2.1 Boolean Algebra

Developed by George Boole in the 19th century, *Boolean Algebra* is an algebraic representation of logic, where 1's denote a "true" value and 0's denote a "false" value. Some common logical operations are described in the tables below:

&	0	1
0	0	0
1	0	1

"and" operator

|	0	1
0	0	1
1	1	1

"or" operator

~	
0	1
1	0

"not" operator

^	0	1
0	0	1
1	1	1

"xor" operator

Note. The "xor" operator stands for "exclusive-or", meaning "either one or the other, but not both".

All of these operations are applied "bitwise" on bit vectors—the jth bit of the result is obtained by applying the operation on the jth bit of the input(s).

Example. *Representing an Manipulating Sets*

Representation

- A width w bit vector can represent a subset of $\{0, \ldots, w-1\}$.
- We do this by letting $a_j = 1$ if j is in our set.
- For example, the bit vector 01101001 would represent the set $\{0, 3, 5, 6\}$. We read from right to left, and we see that the 0th, 3rd, 5th, and 6th entries contain 1's.

Operations

- There are nice parallels between operations on bit vectors and operations on sets, i.e.
  - & on bit vectors is the same as set intersection.
  - | on bit vectors is the same as set union.
  - ^ on bit vectors is the same as the symmetric difference.
  - ~ on bit vectors is the same as taking the complement of a set.

1.2.2 Bit-Level Operations in C

The four boolean operations covered thus far are available in C, and can apply to any "integral data type"(long, int, char, etc). The operators view the arguments as bit vectors and are applied bit-wise.

1.2.3 Logic Operations in C

Note. These are different than the bit-level operations, so don't get them confused.

There are also the logical operators in C, namely `&&`, `||`, `!`.

- These view 0 as "false", and anything non-zero as "true".
- They always return 0 or 1, and can terminate early.

1.2.4 Shift Operations

The shift operator allows you to "move" the bits in a bit vector to give them higher or lower significance.

- The left shift($x \ll y$) shifts the bit vector x to the left by y positions (extra bits on the left are tossed out and empty slots are filled with 0's).
- The right shift($x \gg y$) shifts the bit vector x to the right by y positions (extra bits on the right are tossed out).
  - In a logical shift, the empty slots are filled with 0's.
  - In an arithmetic shift, the empty slots are filled with duplicates of the most significant bit on the left.
- It is undefined behaviour if the shift amount is negative or greater than or equal to the string size.

2 Integers

2.1 Encoding Integers

2.1.1 Bits to Integers

For unsigned integers, we can just use a summation of powers of two to express integers, given by

$$B2U(x) = \sum_{i=0}^{w-1} x_i \cdot 2^i.$$

For signed integers, we use the *Two's complement* form, where we treat the first bit as negative (otherwise known as the "sign bit"), and take the complement of the non-negative bit representation of our integer. It is given by the equation

$$B2T(x) = -x_{w-1} \cdot 2^{w-1} + \sum_{i=0}^{w-2} x_i \cdot 2^i.$$

2.1.2 Numeric Ranges

There's a limited range of numbers that we can represent using our bit representations, because we only use a finite number of bits. For unsigned values, this ranges from

$$\underbrace{000\ldots0}_{0} \quad \text{to} \quad \underbrace{111\ldots1}_{2^w-1}.$$

For Two's complement values, this ranges from

$$\underbrace{100\ldots0}_{-2^{w-1}} \quad \text{to} \quad \underbrace{011\ldots1}_{2^{w-1}-1}.$$

Another interesting number to remember is that $111\ldots1$ represents -1 in Two's complement form.

2.1.3 Values for Different Word Sizes

We have that $|T_{\min}| = T_{\max} + 1$ and $U_{\max} = 2 \cdot T_{\max} + 1$.

In C, the header file `limits.h` declares constant such as:

- `ULONG_MAX`
- `LONG_MAX`
- `LONG_MIN`

> **Note.** The values provided by this header file are platform specific.

2.1.4 Unsigned and Signed Numeric Values

- For non-negative values, unsigned and signed values have the same bit representations.
- Every bit pattern represents a unique integer value, and every representable integer has a unique bit encoding.
- These mappings are invertible, which is to say that $U2B(x) = B2U^{-1}(x)$ and $T2B(x) = B2T^{-1}(x)$.

2.2 Converting and Casting Integers

2.2.1 Mapping Between Signed and Unsigned

To convert between signed and unsigned values, we first convert to the bit representation of the value and then to the other form. In other words,

$$T2U = B2U \circ T2B \quad \text{and} \quad U2T = B2T \circ U2B.$$

We are maintaining the same *bit pattern* when we are doing our mappings.

> **Note.** This does not necessarily preserve the value of the bit pattern, as the same bit pattern does not always mean the same value in Two's complement and unsigned forms.

2.2.2 Relation Between Signed and Unsigned

The main difference when going from unsigned to signed is that the most significant bit goes from being a large positive weight to a large negative weight. Thus the value difference between unsigned and Two's complement is $2^{w-1} + 2^{w-1} = 2^w$ when the most significant digit is a one.

2.2.3 Signed vs. Unsigned in C

- Constants are by default considered to be signed integers, and are only unsigned if explicitly written that way (using a "U" as a suffix, i.e. `4294967259U`).
- Casting
  - Explicit casting between signed and unsigned values is the same as our previously defined functions $U2T$ 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;
```

2.2.4 Casting Surprises

- Expression evaluation
  - If there is a mix of unsigned an signed values in a single expression, *signed values will be implicitly cast to unsigned values.*
  - This includes the comparison operators: `<`, `>`, `==`, `<=`, `>=`

2.2.5 Summary

- When converting, the bit pattern is maintained, but reinterpreted.
- This can have unexpected effects, such as being off by 2^w.

> **Note.** In an expression containing signed and unsigned integers, `int` is cast to `unsigned`.

2.3 Expanding and Truncating Integers

2.3.1 Sign Extension

Given a w-bit signed integer x, we want to convert it to a $w+k$-bit integer with the same value. To do this, we make k copies of the sign bit:

$$X' = \underbrace{x_{w-1} \dots x_{w-1}}_{k \text{ copies of the most significant bit}} x_{w-1} x_{w-2} \dots x_0$$

When converting from a smaller to larger integer data type, C will automatically perform the sign extension.

2.3.2 Truncation

In truncation, bits are removed and the result is reinterpreted. For unsigned integers, this is the same as just performing the modulo operator on the value, and for signed integers it is *similar* to the modulo operator. For small numbers this yields the expected behaviour.

2.4 Adding, Negating, Multiplying, and Shifting

2.4.1 Unsigned Addition

If both of the operands have w bits, then the "true sum" would have to contain $w+1$ bits. The discard carry, denoted as $\text{UAdd}_w(u, v)$, contains w bits (by tossing out the most significant bit).

The standard addition function ignores the carry output, and this is the same as performing modular arithmetic:

$$\text{UAdd}_w(u, v) = u + v \pmod{2^w}$$

2.4.2 Two's Complement Addition

If the sum gets to be too positive or too negative, the sum will overflow in both directions. In other words, if a number gets to be too negative, it will overflow and become a large positive integer. If a number gets to be too positive, it will overflow and become a large (in magnitude) negative integer.

2.4.3 Multiplication

We want to compute the product of two w-bit numbers x and y (which can either be signed or unsigned). However, exact results can be larger than w bits:

- Unsigned multiplication can yield results of up to $2w$ bits, where

$$0 \le x \cdot y \le (2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1.$$

- The minimum for Two's complement multiplication can have up to $2w-1$ bits, where

$$x \cdot y \ge \left(-2^{w-1}\right) \cdot \left(2^{w-1} - 1\right) = -2^{2w-2} + 2^{w-1}.$$

- The maximum for Two's complement multiplication can have up to $2w$ bits, where

$$x \cdot y \le \left(-2^{w-1}\right)^2 = 2^{2w-2}.$$

Thus to maintain exact results, we would need to keep expanding the string size every time we take a product. This can be done in software using "arbitrary precision" packages.

2.4.4 Unsigned Multiplication in C

As mentioned before, the true product of two w-bit strings will (potentially) have $2w$ bits, so C discards the first w bits. We denote this as $\text{UMult}_\text{w}(\text{u}, \text{v})$. Again this is modular arithmetic, so

$$\text{UMult}_\text{w}(\text{u}, \text{v}) = u \cdot v \pmod{2^w}.$$

2.4.5 Signed Multiplication in C

We still ignore the first w bits, but we could have overflowed in either the positive or negative directions. The lower bits are the same.

2.4.6 Power-of-2 Multiplication Using Shifts

Notice that left shifting an bit vector k times yields the same result as if you had multiplied it by 2^k. In other words, $u \ll k = u \cdot 2^k$.

> **Note.** This only works if there is no overflow in the multiplication process.

> **Example.** *Shifting to Multiply*
>
> - `u << 3 == u * 8`
> - `(u << 5) - (u << 3) == u * 24`

Most machines shift and add faster than they can multiply, and the compiler generates this code automatically.

2.4.7 Unsigned Power-of-2 Division Using Shifts

This is done by performing a logical right shift on a vector (replace empty slots with zeroes). We have $u \gg k = \left\lfloor \frac{u}{2^k} \right\rfloor$.

2.4.8 When To Use Unsigned

- Use it when performing modular arithmetic.
- Use it when using bits to represent sets.

> **Note.** Don't use it without fully understanding the consequences, otherwise you can (read: will) have unintended behaviour.

3 Representations in Memory, Pointers, and Strings

Remember that the way that memory is organised in a computer is byte-oriented.

Programs refer to data by address:

- Conceptually, envision memory as a very large array of bytes (not actually true).
- An address is like an index into that array, and a pointer variable stores an address.

> **Note.** The system provides private address spaces to each "process". Thus a program can interact with its own data, but not that of other programs.

3.1 Machine Words

Any given computer has a given "word size", which is the nominal size of integer-valued data. Up until recently, most machines used 32 bits as their word size (limits addresses to 4GB), but nowadays machines have 64-bit word size (up to 18 exabytes of addressable memory). Machines still support multiple data formats, using fractions or multiples of their word size to keep the number of bytes an integer.

3.2 Byte Ordering

There are two main conventions: Big Endian and Little Endian. In the former, the least significant byte has the highest address (more intuitive), and in the latter, the least significant byte has the lowest address.

> **Note.** Big Endian—most to least significant. Little Endian—least to most significant.

3.3 Examining Data Representations

Here is some code that will print out the hexadecimal value at a given pointer location:

```
typedef unsigned char *pointer;
void show_bytes(pointer start, size_t len) {
  size_t i;
  for (i = 0; i < len; i++)
    printf("%p\t0x%.2x\n", start+i, start[i]);
  printf("\n");
}
```

In the above code, `%p` prints a pointer and `%x` prints a hexadecimal.

> **Note.** Different compilers and machines assign different locations to objects. You might even get different results each time the program is run.

3.4 Representing Strings

In C, each string is represented by an array of characters, which are encoded using the ASCII format. In ASCII, the first 7 bits encode the character, while the last bit is 0, which denotes the end of the character. Byte ordering is also not an issue for strings, as endian ordering only matters for individual values, not the array as a whole.