CENG331 - Computer Organization Course Notes

Ozan Şan

October 5, 2019

1 Bits

A common misconception:
$int = \mathbb{Z}$ and, $float = \mathbb{R}$.

For the case of *int*, $x^2 < 0$ can happen.

For the case of *floats*, $(x + y) + z \neq x + (y + z)$ is possible. Commutativity is not guaranteed.

1.1 How is a C code compiled?

1.1.1 Preprocessor

$$\xrightarrow[\texttt{hello.c}]{} \boxed{\text{Preprocessor (cpp)}} \xrightarrow[\texttt{hello.i}]{}$$

In this stage, `#include` and `#define` statements are processed. This happens, for example, for `#include`, a file is copied to the beginning of `hello.c` and `#define` macros are replaced in-place.

1.1.2 Compiler

$$\xrightarrow[\texttt{hello.i}]{} \boxed{\text{Compiler (cc1)}} \xrightarrow[\texttt{hello.s (ASM)}]{}$$

In this stage, the c code without preprocessor stages passes through a compiler and results in an Assembly code block. This does not have the necessary functions in it (for example, `printf`).

1.1.3 Assembler

Here is the assembler in action:

$$\underset{\texttt{hello.s}}{\rightarrow} \quad \boxed{\text{Assembler (as)}} \quad \underset{\texttt{hello.o}}{\rightarrow}$$

In this stage, Assembler takes in the Assembly code, and produces an almost-executable intermediate **Object file**. This **Object file** does not have any externally defined functions in it yet.

1.1.4 Linker

Here is the linker in action:

$$\begin{array}{l} \texttt{printf.o} \searrow \\ \texttt{hello.o} \rightarrow \quad \boxed{\text{Linker}} \quad \rightarrow \texttt{hello} \text{ (executable)} \end{array}$$

At the end, we have an executable in our hands. *Yay.*

1.2 Bit Operations

1.2.1 Representing sets as bits

Take, for example, the set $S_1 = \{0, 3, 5, 6\}$. We can cram this set into a byte, as this: `01101001`. Each bit X_i is set to 1 if $i \in S_1$. As another example, let's take $S_2 = \{0, 2, 4, 6\}$ and its corresponding bit representation, `01010101`.

Now, we can use mathematical operations on these bit representations provided by the processor:

Bit Operation	Mathematical Operation	Bit Result	Set result
&	`Intersection` ($\bigcap$)	`01000001`	$\{0, 6\}$
$\vert$	`Union` ($\bigcup$)	`01111101`	$\{0, 2, 3, 4, 5, 6\}$
$\wedge$	`Symmetric Difference`	`00111100`	$\{2, 3, 4, 5\}$
$\sim$	`Complement`	`10010110` (On S_1)	$\{1, 2, 4, 7\}$

An important point to make is that these bitwise operations are not the same as logical operators (for example, `&&`) as logical operators operate on the

principle that everything non-zero can be treated as `True`, and this means we will not get a result beyond the first bit (as the result will only be either 0, or 1). Also, logical operators can terminate early for making computations faster. (`!!0x41` $\neq$ `0x41`, but, `!!0x41 = 0x01 (True)`)
An example usage of the bitwise operations is below. We will try to write a `void swap(int* a, int* b)` function without an additional variable. We can achieve this by using XOR.

```c
void swap(int* a, int* b)
{
        *a = *a ^ *b; //1
        *b = *a ^ *b; //2
        *a = *a ^ *b; //3
}
```

Here's what happens when we execute this, line by line:

Line	Value of A	Value of B
1	$A \oplus B$	B
2	$A \oplus B$	$B \oplus B \oplus A = A$
3	$A \oplus B \oplus A = B$	A
End	B	A

With bit operations, we have saved on some storage. *Yay.*

1.2.2 Shifting

We can shift the bits in a variable. Left shift operations replace the missing bits by 0's, whereas right shift operations may or may not do that, depending on the Most Significant Bit (this is done for not losing the sign information located mainly on MSB).

Operation	01100010	10100010
$<< 3$	00010000	00010000
Log. $>> 2$	00011000	00101000
Arithmetic $>> 2$	00011000	11101000

1.3 Encoding integers

We define two functions to interpret the meaning of a binary variable x depending on the type (whether it is `unsigned` or not).
For `unsigned`:

$$B2U_w(x) = \sum_{i=0}^{w-1} x_i 2^i$$

For signed `int` (will be called as Two's Complement, or "T"):

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