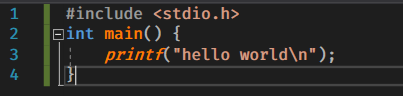
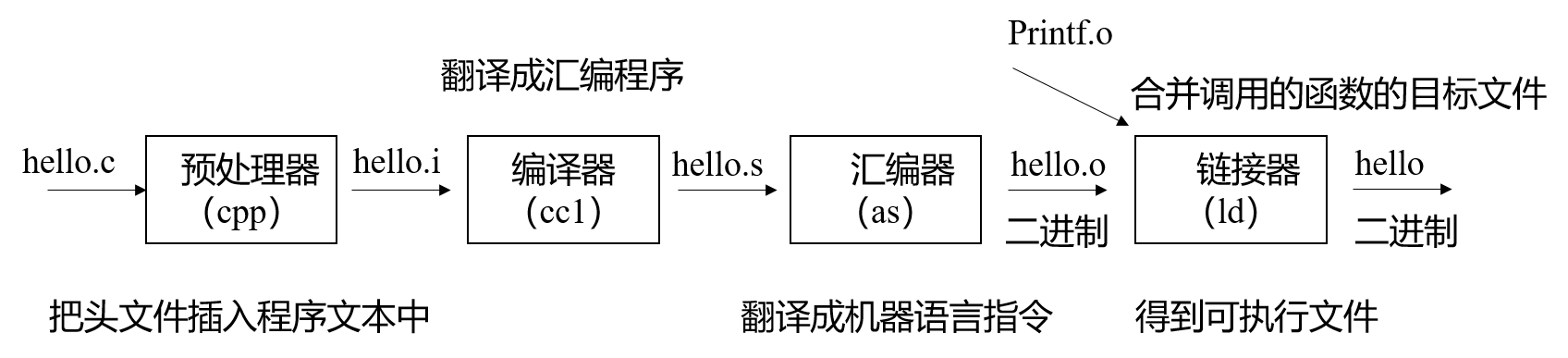
总线是贯穿整个系统的一组电子管道，负责在各个部件之间传递信息字节。通常总线传送定长的字节块，即*字*  
*字*中的字节数就是*字长*  
因为一个字节有8bit，即8位，所以我们所说的  
*32位*系统就是字长为4；  
*64位*系统就是字长为8；  
每台计算机都有一个*字长*，指明整数和指针数据的大小。字长决定虚拟地址空间的最大大小，对于字长为w位的机器，虚拟地址的范围为  
0 − 2 w − 1 0-2^{w-1} 0−2*w*−1  
每个字节都有它对应的地址。

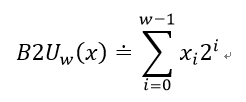


  
hello程序的生命周期  
编译器将源程序文件hello.c翻译成可执行目标文件hello，分为四个阶段  
1.预处理阶段  
2.编译阶段  
3.汇编阶段  
4.链接阶段

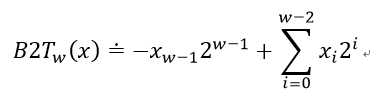
**整数表示**

**无符号数**

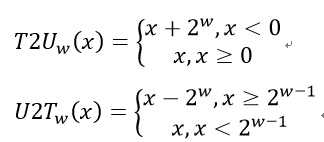
将整数的位向量表示为  
在这里插入图片描述

则其无符号表示为  


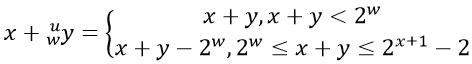
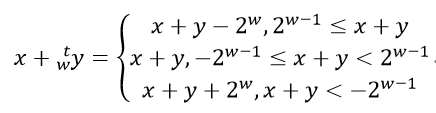
**有符号数**

补码形式为  


**无符号数与有符号数之间的转换**

  
值可能会变，位模式保持不变。  
C语言执行一个运算时，如果一个运算数无符号，另一个有，那么会隐式地将有符号转为无符号，有些情况下会导致结果错误。  
避免这个错误的方法就是不使用无符号数。

*算数右移*：左边全部补最高有效位  
*逻辑右移*：左边全部补0

无符号加法：  
有符号加法：  


*//判断无符号加法O不ok,有溢出就返回false*

bool uadd\_ok(unsigned int x, unsigned int y) {

unsigned sum = x + y;

return sum >= x;

}

*//判断有符号加法O不ok,有溢出就返回false*

bool tadd\_ok(int x, int y) {

int sum = x + y;

bool neg\_over = x < 0 && y < 0 && sum >= 0;

bool pos\_over = x >= 0 && y >= 0 && sum < 0;

return !neg\_over && !pos\_over;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12

为什么会发生溢出，举个简单的例子，假设一个4位无符号整数，其范围为0~15；如果要算15+15，我们来看看会发生什么：  
1111+1111=11110；但是只有4位，所以最高位要被丢掉，得到的结果就是1110=14。  
假设一个4位有符号整数，其范围为1000~0111=-8 ~7；如果要算7+7，我们来看看会发生什么：  
0111+0111=1110=-2；  
咦？咋回事，15+15不应该是30吗？7+7不应该是14吗？没错，计算机算术不一定都是对的，当溢出发生时，结果简直是大错特错。  
而这种溢出往往不会得到编译器的提醒，所以检测溢出十分有必要，检测无符号、有符号加法的函数如上