HW1/H1问题回答

问题1-1

如果在命令行下执行`gcc -DNEG -E sample.c -o sample.i`生成的`sample.i`与之前的有何区别?

区别在于`int a = 4;`一句会变成`int a = -4`, 产生这个区别的原因在于编译时增加了编译选项`-DNEG`, 即定义了`NEG`, 因而程序中的`M`在预编译阶段会被展开为`-4`.

问题1-2

请对比`sample-32.s`和`sample.s`， 找出它们的区别， 并上网检索给出产生这些区别的原因。

32位	64位	原因
pushl movl	pushq movq	后缀`l`表示`long`,而`q`表示`quad`, 在32/ 64位系统中因使用寻址能力的不同, 导致需要使用不同位数寄存器来做地址操作, 进而导致指令也应该不同
%esp等	%rsp等	%esp等寄存器表示的是32位的寄存器, 而%rsp等表示的是64位的寄存器, 由于编译时指定位数不同, 则寻址能力的不同会导致最后使用的寄存器位数不同.
leave	popq %rbp	这里实际上实现的功能都是相同的, 实现了退栈操作, 以实现函数的返回.

参考资料:

- X86 Assembly/GAS Syntax

问题1-3

你可以用`clang`替换`gcc`， 重复上面的各步， 比较使用`clang`和`gcc`分别输出的结果有何异同。

- 预处理: 对于`#include`的头文件, 其前后数字标号略有不同. 这一串字符串被称为`linemarkers`. 当编译器遇到错误时, 有助于报错的定位, 并且可以作为debug的信息使用.
    - 对于`gcc`: 它们的含义是`# linenum filename flags`, 其中`flags`的含义有:
        - 1: 指示新文件的开始
        - 2: 指示返回到一个文件
        - 3: 指示接下来的代码来自系统头文件
        - 4: 指示接下来的代码应该被认为包装在了一个`implicit extern "C" block`
    - 对于`clang`也类似.
- 编译:
    - 指令顺序略有不同
    - 运算方式不同: `gcc`会直接使用一些能够直接操作内存地址上数据的指令; 而`clang`里似乎有的会使用先`mov`到寄存器, 然后操作, 再`mov`回内存的情况.
    - 初始化寄存器为0的指令不同: `gcc`直接将%0的值赋给寄存器; 而`clang`则使用寄存器自己与自己作`xor`得到0.
    - 是否使用`leave`: `gcc`在32位下, 使用了leave; 而`clang`没有.
    - 其余都是一些小细节诸如注释不同, 标签不同等等, 在此不再赘述.
- 反汇编: 反汇编的过程都是使用`objdump`对`.o`文件做反汇编, 因此结果与前面编译结果相关, 前面编译的不同, 在反汇编中也同样体现出来了.

参考资料:

- What are these unfamiliar lines in the C preprocessed file?
- 9 Preprocessor Output