Homework Problems

*CSAPP 2ed Chapter 3*

### 3.54

本题考察对instructions的理解，要求根据ASM写出对应的C代码。

### 3.55

本题考察对mov和arithmetic instructions的理解，以及和C代码的对应关系。

我要做两件事情：描述ASM的算法、添加注释。

算法：先对y进行sign extended。然后依据下列公式，针对x与y 的不同部分计算乘积，然后组合结果，等价于x\*y。

|  |  |  |
| --- | --- | --- |
|  | 64 bit | 32 bit |
|  |  |  |
| \* |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

结合以上公式，给出注释：

|  |  |
| --- | --- |
|  |  |
| 1 | 获取 |
| 2 | 获取y |
| 3 | 将y保存到%edx |
| 4 | 对%edx做arithmentic right shift，得到 |
| 5 | 移动 |
| 6 | 计算 |
| 7 | 获取 |
| 8 | 计算 |
| 9 | 计算 |
| 10 | 计算 |
| 11 | 计算 |
| 12 | 获取dest的值 |
| 13 | 将保存到memory |
| 14 | 将保存到memory |

### 3.56

本题考察对conditional instructions的理解。

基本性质：

1. 值不变的register，对应C中的不变量
2. test指令的operand对应循环变量
3. %eax值的来源，对应返回变量
4. 根据ASM的1、2行，容易确定x对应%esi，y对应%ebx。根据第3条性质，以及ASM第13行，可以确定result对应%edi。根据第2条性质，以及ASM第11行和C第5行，可以确定mask对应%edx。
5. result值保存在%edi中，根据ASM第3行，初始值是1431655765；

result值保存在%edi中，根据ASM第3行，初始值是1431655765；

1. 根据ASM第11、12行，可知test condition是mask<=0
2. 在ASM的循环主体中，只有第10行改变了%edx的值。结合第9、10两行，mask的update方式是

mask = mask >> ((char) n);

1. 在ASM的循环主体中，只有第8行改变了%edi的值。结合第6、7、8行，result的update方式是

result ^= x & mask;

1. 见代码。

### 3.57

本题考察对conditional data transfer的理解。

基本性质

* 即使是if-else语句，也可以编译成conditional data transfer

至于本题，完全不理解如何解答。

### 3.58

本题与p.249的例子相似。考察对switch语句的理解。

本题虽说写出了代码，但编译后的ASM与题中给出的不同，因此不知是否正确。

### 3.59

本题仍是考察对switch语句编译结果的理解。

基本性质

* ASM包括了所有cases的执行过程
* Jump table涵盖了所有cases初始instructions的地址

基本方法

* 先假设所有cases互相独立，根据ASM写出代码
* 再结合ASM，将有交集的cases合并，更新代码

虽然写了一个实现，但是编译后，再反汇编的结果与题中给出的ASM有差异。不知问题在哪。

### 3.60

本题考察多维数组元素的分布规律。

基本性质

* 多维数组的元素实际存储为一维数组，用计算下标的方式表示多维性

基本方法

* 计算地址：下标转换成地址
* 确定数组维度：
  + 根据元素地址的计算方式，可以S、T
  + 再结合数组的size，可知R

1. 注释

|  |  |
| --- | --- |
|  |  |
| 1 | 参数i |
| 2 | 参数j |
| 3 | 9\*j |
| 4、5、6 | (i<<6)-i = i\*63 |
| 7 | 9\*j+ i\*63 |
| 8 | k+9\*j+ i\*63 |
| 9 | 获取A[i][j][k] |
| 10 | 参数dest |
| 11 | 将A[i][j][k]保存到\*dest |
| 12 | 返回size=2772 |

根据第8、9行，结合(1)的公式，可知

因此，

根据第12行，可知A的size为2772 bytes。结合size的计算公式：，可知

于是，R=1。

### 3.61

本题考察二维动态数组的使用、以及代码优化。

基本性质

* 可以将多维数组看成一维数组

基本方法

* 没有想法。。。。。

### 3.62

本题考察二维动态数组元素在内存中的布局。

基本性质

* 循环主体表现为，变量在数组元素上的不断转移

基本方法

* 利用多维数组元素地址的计算公式

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

1. 根据第7行，可知M=76/4=19.
2. 根据第5行，结合C代码，可知循环变量j的值保存在%ecx中；根据第8行，可知该行对应C中的条件判断，因此i保存在%edi中。
3. 在新的实现中，用地址计算替代数组索引。具体见代码transpose\_opt.c。本小题的额外价值是，我应用了在2.70问题学习到的经验，加深了我对程序设计方法的理解。

### 3.63

本题要从ASM分析数组的元素数量。ASM中的关键部分，是循环主体的实现。其中，循环变量的判断条件、指针的移动，是分析数组元素数量的关键。

分析.L4循环主主体，可知第18行实现了循环变量的判断。根据第16行，可知%edx中的值是循环变量，因此，。结合第1、2、4行，可知。综上，。

观察第15行，应是实现了将数组元素累加到结果变量上。第17行，应是移动指针，指向同一列的下个元素。可知，%ebx的值，是同一列的两个相邻元素内存地址的差值，等于。

相应地，。

### 3.64

本题关注：当函数的参数为struct类型变量时，如何在函数之间传递参数。

1. 根据第10、12行，可知%eax中保存着返回值的地址。根据第8行，可知%edx中保存着s1的p字段；根据第11行，%ebx保存着s1的v字段。
2. Stack frame structure

|  |  |
| --- | --- |
| 0 | …… |
| -4, %ebp | Saved %ebp |
| -8 | s2.sum |
| -12 | s2.prod |
| -16 | s1.v |
| -20, %esp | s1.p |

根据word\_sum的第9、10、11、12行，确定s2的布局；根据diff的第9、10行，确定s1的布局。

1. 将字段作为独立参数，分布在stack中传递。
2. 当返回值是struct时，在被调用函数中直接修改调用函数中的结构体。也就是直接修改caller中的数据，就不需要传递额外的返回值。

### 3.65

本题结合了struct参数传递、多维数组。

根据ASM，q和p在内存中的布局如下

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| q: |  |  | p: |  |
| 0 | array |  | 0 | x |
| 4 |  |  | 4 |  |
| 8 | t |  | 8 |  |
| 12 | s |  | 12 |  |
| 16 |  |  | 16 |  |
| 20 |  |  | 20 |  |
| 24 |  |  | 24 |  |
| 28 | u |  | 28 |  |
| 32 |  |  | 32 |  |
| 36 |  |  | 36 |  |
| 40 |  |  | 40 |  |
| 44 |  |  | 44 | y |
| 48 |  |  | 48 |  |

数组array的元素类型是char，size为1 byte，共用了8 bytes，因此数量为8，即B为8。

同样的，数组s的元素类型是short，size为2 bytes，共用了16 bytes，因此数量为8，即B为8。

接下来，分析p->x。数组x的元素类型是short，size为2 bytes。数量为8\*A，因此，共占用16\*A bytes。而根据ASM第5行，x共占用44 bytes。？？？为什么是44，不是48？？？

如果是48，那么A恰好为3。

### 3.66

本题考察struct类型对象在内存中的布局以及在函数中的使用。

|  |  |
| --- | --- |
| b\_struct |  |
|  | left |
| a[0] | idx |
|  | x[0] |
|  | x[1] |
|  | … |
| a[1] | idx |
|  | x[0] |
|  | x[1] |
|  | … |
|  | … |
|  | … |
|  | right |

注释：

|  |  |
| --- | --- |
|  |  |
| 1 |  |
| 2 |  |
| 3 |  |
| 4 |  |
| 5 | 获取i |
| 6 | 获取bp |
| 7 | 0x1c\*i |
| 8 | 8\*i |
| 9 | 8\*i-i=7\*i |
| 10 | M[bp+0x1c\*i+0x4] = ap->idx + 7\*i |
| 11 | right |
| 12 | left |
| 13 | right+left -> M[bp+4\*(ap->idx + 7\*i)+0x8] |
| 14 |  |
| 15 |  |
| 16 |  |

1. 结合上面b\_struct的布局，并根据第7、10行，可知a\_struct的size是0x1c；根据第11、12行，可知数组b\_struct->a的size是。因此，
2. 根据第13行，可知a\_struct中包括7个size为4 bytes的元素。因此，推断idx是一个int；x是包含6个int元素的数组。

### 3.67

本题考察struct类型对象在内存中的布局。

基本性质

* Union的size由其中size最大的元素决定
* 所有元素不会同时出现在union中

1. Offset

|  |  |
| --- | --- |
|  | Offset |
| e1.p | 0 |
| e1.x | 4 |
| e2.y | 0 |
| e2.next | 4 |

1. 根据性质2，ele的size为8 bytes。
2. 本小题中有三样事物：computation、ASM、C code。可行的解法是：先根据ASM，结合memory抽象模型，用图形描述computation，再用C表示computation，得到相应的C表达式。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| up-> |  |  |  |  |  |  |
|  |  | <- B |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  | => %ecx | e1.p | -> |  | <- A |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  | e1.x |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

核心计算：e1.x=A-B.

附注：结构体元素和指针的引用要注意。

### 3.68

本题考察字符串数组的布局、内存的动态分配。

由于字符串的长度未知，我的策略是用链表，将字符串分到多个节点上。

具体代码见good\_echo.c。

附注：此外，还有其它方法吗？题干中提到了fgets函数，如何派上用场？

### 3.69

本题考察struct类型。

题目中包括3个部分：数据类型的定义、函数原型、ASM。要求根据以上信息，给出函数的定义。方法上，主要是根据ASM，分析Computation，再用C表示。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| tp -> | left | -> | left | -> | …… |
|  | right |  | right |  |  |
|  | value | -> %rax | value | -> %rax |  |

通过分析，我认为trace函数将返回binary tree最左端的元素值。

具体见代码trace.c。

### 3.70

本题考察struct类型，是在3.69基础上更深入一步，加入递归调用、CMOV。

|  |  |  |
| --- | --- | --- |
|  | |  |
| 1 | traverse | |
| 2 | | Preparation |
| 3 | |  |
| 4 | |  |
| 5 | |  |
| 6 | | Result=Max |
| 7 | | tp |
| 8 | | 检查tp |
| 9 | | if tp is NULL, goto the end |
| 10 | |  |
| 11 | |  |
| 12 | | 以tp->left为参数，递归调用traverse |
| 13 | | 将结果放到%r12 |
| 14 | |  |
| 15 | | 以tp->right为参数，递归调用traverse，结果放在%rax |
| 16 | | 比较两次递归调用的结果 |
| 17 | | 取其中较小值 |
| 18 | | 在用较小值与tp->val比较 |
| 19 | | 取其中更小值 |
| 20 | .L9 | |
| 21 | | 恢复stack |
| 22 | |  |
| 23 | |  |
| 24 | |  |
| 25 | | 将最小值做为结果返回 |

根据以上分析，trace函数将遍历binary tree，返回其中最小值。代码见traverse.c。