计算机系统结构(A)

第 1 次作业

Log Creative

2021 年 6 月 27 日

1. 假定机器M的时钟频率为1.2GHz，某程序P在机器M上的执行时间为12秒钟。对P优化时，将其所有的乘4指令都换成了一条左移2位的指令，得到优化后的程序P'。已知在M上乘法指令的CPI为5，左移指令的CPI为2，P的执行时间是P'执行时间的1.2倍，则P中有多少条乘法指令被替换成了左移指令被执行？

解. 根据公式：

$$\text{CPU 时间} = \text{指令数} \times \frac{\text{CPI}}{\text{时钟频率}}$$

可以得到对于程序P，

$$12\text{s} = n \times \frac{5}{1.2\text{GHz}}$$

对于程序P'，

$$10\text{s} = (n-w) \times \frac{5}{1.2\text{GHz}} + w \times \frac{2}{1.2\text{GHz}}$$

联立可以解得

$$w = 8 \times 10^8$$

条乘法指令被替换成了左移指令被执行。

2. 图形处理器中经常需要的一种转换是求平方根。浮点（FP）平方根的实现在性能方面有很大差异，特别是在为图形设计的处理器中，尤为明显。假设FP平方根（FPSQR）占用一项关键图形基准测试中30%的执行时间。有一项提议：升级FPSQR硬件，使这一运算速度提高到原来的10倍。另一项提议是让图形处理器中所有FP指令的运行速度提高到原来的1.6倍，FP指令占用该应用程序一半的执行时间。设计团队相信，他们使所有FP指令执行速度提高到1.6倍所需要的工作量与加快平方根运算的工作量相同。试比较这两种设计方案。

解. 根据 Amdahl 定律：

$$\text{改进后的运行时间} = \frac{\text{受改进影响的执行时间}}{\text{改进量}} + \text{不受影响的执行时间}$$

则假设该基准测试原来需要运行 100 秒，则升级 FPSQR 硬件后运行时间变为

$$73 = \frac{30}{10} + (100 - 30)$$

而使所有 FP 指令的运行速度提高到原来的 10 倍后的运行时间变为

$$81.25 = \frac{50}{1.6} + (100 - 50)$$

因此如果两种方案的工作量是相同的，那么**升级 FPSQR 硬件**会让执行时间更短，该方案是更好的。

3. 假设我们在对有符号值使用补码运算的32位机器上运行代码。对于有符号值使用的是算术右移，对无符号值使用的是逻辑右移。变量的声明和初始化如下：

```
int x = foo(); //调用某某函数，给x赋值
int y = bar(); //调用某某函数，给y赋值
unsigned ux = x;
unsigned uy = y;
```

对于下面每个表达式，证明对于所有的x和y值，都为真（等于1）；或者（2）给出使得它为假的x和y值；

解.

A. `(x>0) || (x-1<0)` **假:** $x \geq 1$ 或 $x \leq 0$ 都会使其为假。

B. `(x&7) != 7 || (x<<29<0)` **真。** 因为如果第一个式子为假就意味着`x&7 == 7`，而7是`111`，所以按位与运算后为7就意味着x的后三位为`111`。左移29位后x一定是`1110 0000 0000 0000 0000 0000 0000 0000`=-536870912，由于第一位为符号位，所以该值必定为负数，所以后者成立。

C. `(x * x) >= 0` **假。** 因为32位符号数的最大值为`0111 1111 1111 1111 1111 1111 1111 1111`$= 2^{31} - 1$，那么只要$x^2 \geq 2^{31}$就会导致上溢，$x \geq 46341$，上溢出就会导致结果可能会小于0，其中$x = 46341$就会使得其为负值。所有使得该结果为负的情况是`(x * x)& 0x8000 0000 == 0x8000 0000`的x，也就是截断后的结果最高位为1的情形。

D. `x>0 || -x>=0` **假：** 当x =`1000 0000 0000 0000 0000 0000 0000 0000`时，前者不成立因为符号位为 1，而后者采用加法逆元定义，由于`0x8000 0000` + `0x8000 0000` = `0x0000 0000`被截断，所以其最小值是其本身，也就是仍然是一个小于 0 的数字。

E. `x+y == uy+ux` **真。** 符号整数和无符号整数在二进制上的加法规则是一致的，转换为无符号整数时由于宽度是相等的，所以数据不会有舍弃，得到的二进制码也将是一致的。所以该式永真。

F. `x*~y + uy*ux == -x` **真。** `x*~y + uy*ux = x*(-(y+1)) + uy * ux = -x`，乘法运算两者同用一个乘法器，没有二进制码上的区别。

G. `x*4 + y*8 == (x<<2)+(y<<3)` **真**。左移运算是等价的，`x*4==x<<2`，`y*8==y<<3`。

H. `((x>>2)<<2)<=x` **真**。算数右移不会影响符号位的信息，左移后可能会导致后面两位为0，所以正数经过该操作后会变为更小的正数，负数经过该操作后会变为更小的负数。

4. 假定在一个程序中定义了变量x、y和i，其中，x和y是float型变量（用IEEE754单精度浮点数表示），i是16位short型变量（用补码表示）。程序执行到某一时刻，x = –0.125、y=7.5、i=100，它们都被写到了主存（按字节编址），其地址分别是100，108和112。请分别画出在大端机器和小端机器上变量x、y和i在内存的存放位置。

解. 规格化浮点数 $x = -0.125 = -\frac{1}{2^8} = -0.001_2 = -1 \times 2^{-3}$，$y = 7.5 = 111.1_2 = 1.111 \times 2^2$，$i = 01000100_2$。浮点数按照 IEEE754 表示：

x: 1 | 0 1 1 1 1 1 0 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

x: `0xBE000000`

y: 0 | 1 0 0 0 0 0 0 1 | 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

y: `0x40F00000`

i: 0 0 0 0 | 0 0 0 0 | 0 1 0 0 | 0 1 0 0

i: `0x0044`

内存地址	100	104	108	112
大端机器	BE 00 00 00	Other Data	40 F0 00 00	00 44
小端机器	00 00 00 BE	Other Data	00 00 F0 40	44 00

5. We are running programs on a machine with the following characteristics:

- Values of type int are 32 bits. They are represented in two’s complement, and they are right shifted arithmetically. Values of type unsigned are 32 bits.
- Values of type float are represented using the 32-bit IEEE floating point format, while values of type double use the 64-bit IEEE floating point format.
- We generate arbitrary values x, y, and z, and convert them to other forms as follows:

```
/* Create some arbitrary values */
int x = random();
int y = random();
int z = random();
 /* Convert to other forms */
unsigned ux = (unsigned) x;
unsigned uy = (unsigned) y;
double dx = (double) x;
double dy = (double) y;
double dz = (double) z;
```

For each of the following C expressions, you are to indicate whether or not the expression always yields 1. If so, circle "Y". If not, circle "N" and tell why.

Expression	Always True?	Why?
`(x<y)==(-x>-y)`	Y Ⓝ	最小的整型数其加法逆元仍为其本身，仍然是最小的整型数，在前者可能会真的时候，后者为假，这样整体会变成假。
`((x+y)<<4 + y-x == 17*y + 15*x)`	Ⓨ N	`(x+y)<<4+y-x = 16*x+16*y+y-x = 17*x-15*x` 不影响运算结果。
`∼x+∼y+1==∼(x+y)`	Ⓨ N	`∼x+∼y+1 = (-(x+1))+(-(y+1))+1 = -(x+y)+1 = ∼(x+y)` 运算结果是不影响的。
`ux-uy==-(y-x)`	Ⓨ N	不影响运算结果，因为使用同一套加法减法器。
`(x>=0)||(x<ux)`	Ⓨ N	$x < 0$时，整型数的最高位必定为 1，无符号整型数必定为很大的正数，所以后者成立。
`((x>>1)<<1)<=x`	Ⓨ N	左边运算的结果是最后一位被替换成了0，那么这个记过一定是小于原来的数的，不论正负。
`(double)(float)x==(double)x`	Y Ⓝ	先转换成 float 会有舍入现象，可能会导致不相等。
`dx + dy == (double) (y+x)`	Y Ⓝ	int 的两值相加可能会导致上溢，转换成 double 这个更大范围的量也会导致截断。
`dx + dy + dz == dz + dy + dx`	Y Ⓝ	浮点数加法不符合结合律。
`dx * dy * dz == dz * dy * dx`	Y Ⓝ	浮点数乘法不符合结合律。