作业1： Amdahl定律、数的表示

520030910281 肖真然

一、 思考题

要求：下面的子问题需要书面完成，可以手写后扫描、也可以直接编辑本文件回答问题。

1、假定某个程序的代码可并行化部分是98%，提供给你多个处理器核(core)来将问题并行化。如果并行化后加速比要达到7以上，至少需要多少处理器核将问题并行化处理。

解：

$$Speedup = \frac{T(1-f)+Tf}{T(1-f)+\frac{Tf}{N}} = \frac{1}{(1-f)+\frac{f}{N}} = \frac{1}{(1-0.98)+\frac{0.98}{N}} \geq 7$$

解得$N \geq \frac{7\times 0.98}{1-7\times(1-0.98)} \approx 8$，故至少需要8个处理器核将问题并行化处理才能使加速比达到7以上。

2、以下是两段 C 语言代码，函数 arith( )是直接用 C 语言写的，而 optarith( )是对 arith( )函数以某个确定的 M 和 N 编译生成的机器代码反编译生成的。根据 optarith( )，可以推断函数 arith( ) 中 M 和 N 的值各是多少？

```
#define    M
#define    N
int    arith      (int x, int y)
{
       int    result = 0 ;
       result = x*M + y/N;
       return result;
}

int optarith ( int x,    int y)
{
       int    t = x;
       x <<= 2;
       x -= t;
       if ( y < 0 )    y += 15;
       y>>=4;
       return x+y;
 }
```

解： M = 3， N = 16.

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 C 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；

A. (x>0) || (x-1<0)

B. (x * x) >= 0

C. x<0 || -x<=0

D. x>0 || -x>=0

E. x+y == uy+ux

F. x*~y + uy*ux == -x

G. x*4 + y*8 == (x<<2)+(y<<3)

H. ((x>>2)<<2)<=x

解：

A. 当 x≤0 且 x-1≥0 时为假，即 x=1000 0000…（31 个 0），x-1=0111 1111 …（32 个 1）时为假。

B. 当 x = 50000 时，x*x 的结果的 32 位二进制码为 1001 0101 0000 0010 1111 1001 0000 0000，符号位为 1，故 x*x≥0 为假。

C. 当 x＜0 时，原式为真；当 x≥0 时，x 符号位为 0，则-x 要么符号位为 1，要么-x=0，故-x≤0，原式为真。故原式永真。

D. 当 x=-2^31 时，x＜0，-x＜0，原式为假。

E. 因为 x 与 ux，y 与 uy 在物理层面上编码相同，且有符号数与无符号数共用一个加法器，所以原式永真。

F. x*~y=x*(-y-1)+ux*uy=x*(-y)-x+ux*uy=-x，故原式永真。

G. 计算机内部编译器会将乘 2^k 解释为左移，所以原式永真。

H. 右移后再左移对高位无影响，对于低位，若右移损失的低位全为 0，则再左移后所得结果还是与原 x 相等，若右移损失的低位中有 1，则再左移后所得结果小于原 x，故原式永真。

4、给定：

- int 、unsigned int 长度为 32 bits.
- float 是 32 位 IEEE 754 单精度浮点数，double 是 64 位 IEEE 754 双精度浮点数
- 变量之间的装换如下：

```
/* 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;

以下表达式，哪些恒定为 true？是的话圈 “Y”，不是的话圈“N” ，并指出原因，给出反例。

Expression	Always True?
`(x<y) == (-x>-y)`	Y N
`((x+y)<<4) + y-x == 17*y+15*x`	Y N
`~x+~y+1 == ~(x+y)`	Y N
`ux-uy == -(y-x)`	Y N
`(x >= 0) || (x < ux)`	Y N
`((x >> 1) << 1) <= x`	Y N
`(double)(float) x == (double) x`	Y N
`dx + dy == (double) (y+x)`	Y N
`dx + dy + dz == dz + dy + dx`	Y N
`dx * dy * dz == dz * dy * dx`	Y N

解：

1. N，当 x=2^31-1，y=-2^31 时，原式左值为 0，右值为 1，原式为假。

2. Y，乘法与左移等价。

3. Y，[x]补 + [y]补 = [x + y]补 等价于 ~x + 1 + ~y + 1 = ~(x + y) + 1 等价于原式。

4. Y，无符号数与有符号数在内存中编码一致且共用一套加法器。

5. N，当 x<0 时，原式左值为 0，右值为 0，原式为假。

6. Y，右移后再左移对高位无影响，对于低位，若右移损失的低位全为 0，则再左移后所得结果还是与原 x 相等，若右移损失的低位中有 1，则再左移后所得结果小于原 x。

7. N，当 x=2^31-1 时，原式为假。因为转换为 float 有精度损失。

8. N，当 x=2^31-1，y=2^31-1 时，原式为假。因为 double 类型的位数更多，dx+dy 不会溢出，而 y+x 可能溢出，溢出后再转换成 double 类型就与 dx+dy 不一样了。

9. Y，double 类型位数多，做加法不会溢出。

10. N，当 x = 2147483647，y = 2199999999，z = 2148888888 时，原式为假。因为乘法中间可能会有溢出与舍入，所以交换顺序后溢出与舍入可能不一样。计算机输出结果：左值=9655009342644796883 右值=9655009342644785888

二、 实践题：位级运算、数的编码

要求：不需要提交代码，只需要在报告中，把你的运行结果、以及你实现的五个函数的源代码贴上来。

先下载 datalab.tar 文件
#解压缩：
$ tar -xvf datalab.tar
#进入文件夹
$ cd datalab
查看文件夹下的文件
$ ls
执行命令
$ make clean
$ make all
运行：
$ ./fshow 一个浮点数例如 34.5
你可以看到这个数的 float 机器编码
$ ./ishow 一个整数例如 20
你可以看到这个数的十六进制的机器编码
$ ./btest
你会看到程序错误的提示：

```
Score   Rating  Errors  Function
ERROR: Test allOddBits(-2147483648[0x80000000]) failed...
...Gives 2[0x2]. Should be 0[0x0]
ERROR: Test isLessOrEqual(-2147483648[0x80000000],-2147483648[0x80000000]) faile
d...
...Gives 2[0x2]. Should be 1[0x1]
ERROR: Test logicalNeg(-2147483648[0x80000000]) failed...
...Gives 2[0x2]. Should be 0[0x0]
ERROR: Test floatScale2(0[0x0]) failed...
...Gives 2[0x2]. Should be 0[0x0]
ERROR: Test floatFloat2Int(0[0x0]) failed...
...Gives 2[0x2]. Should be 0[0x0]
Total points: 0/20
```

你需要做的是：修改 bit.c 文件中的几个函数，完成规定的功能，仔细阅读 bit.c 中各函数前的注释，了解各函数应该能达到的功能。这些函数有：

```
// 判断整数 x 的所有奇数位是否都为 1
// 可以使用的运算符：~ & ^ | + << >>,运算符最多可以用多少次参见程序注释
int allOddBits(int x) {
  return 2;
```

```
}

//使用位级运算符实现判断整数 x<= y
//可以使用的运算符：~ & ^ | + << >>,运算符最多可以用多少次参见程序注释
int isLessOrEqual(int x, int y) {
   return 2;
}

//使用位级运算求逻辑非！
//可以使用的运算符：~ & ^ | + << >>,运算符最多可以用多少次参见程序注释
int logicalNeg(int x) {
   return 2;
}

//求 2 乘一个浮点数，
//可使用任意整数的合法运算符，例如: &, |, ^ , ||, &&, +，  -，   if, while
//运算符最多可以用多少次参见程序注释
unsigned floatScale2(unsigned uf) {
   return 2;
}

// 将浮点数 uf 转换为整数，返回其 32 位的位级表达
// 可使用任意整数的合法运算符，例如: &, |, ^ , ||, &&, +，  -，   if, while
// 运算符最多可以用多少次参见程序注释
int floatFloat2Int(unsigned uf)
 {
   return 2;
}
```

关于浮点数的编码，[IEEE-754 Floating Point Converter (h-schmidt.net)](#) 这里有一个转换器，希望对你有帮助。

修改程序后重新编译：

$ make clean

$ make all

执行：

$ ./btest

如果你的实现全部正确，应该得到以下结果，这 5 个函数的得分情况如下，满分为 20 分。

```
Score   Rating  Errors  Function
 2      2       0       allOddBits
 4      4       0       isLessOrEqual
 4      4       0       logicalNeg
 5      5       0       floatScale2
 5      5       0       floatFloat2Int
Total points: 20/20
```

如果有扣分，说明函数实现没有符合要求，使用了不允许使用的运算符。使用命令

$ ./dlc bits.c

可以调用文件包中提供的规则检查器，检查哪个运算符是不合规定的。

注：本实验选自 CMU CSAPP，如果你想了解 CMU CSAPP：Datalab 的完整要求，可以查看：http://csapp.cs.cmu.edu/3e/labs.html 找到相关的文档和代码。

运行结果：

```
thousanrance@thousanrance-VirtualBox:~/Desktop/Code/SS/hw1/datalab$ ./btest
Score   Rating  Errors  Function
 2      2       0       allOddBits
 4      4       0       isLessOrEqual
 4      4       0       logicalNeg
 5      5       0       floatScale2
 5      5       0       floatFloat2Int
Total points: 20/20
thousanrance@thousanrance-VirtualBox:~/Desktop/Code/SS/hw1/datalab$
```

源代码：

Open ▾ bits.c ~/Desktop/Code/SS/hw1/datalab Save

```
138 //1
139 /*
140  * allOddBits - return 1 if all odd-numbered bits in word set to 1
141  *   where bits are numbered from 0 (least significant) to 31 (most significant)
142  *   Examples allOddBits(0xFFFFFFFD) = 0, allOddBits(0xAAAAAAAA) = 1
143  *   Legal ops: ! ~ & ^ | + << >>
144  *   Max ops: 12
145  *   Rating: 2
146  */
147 int allOddBits(int x)
148 {
149 //your codes Here
150         int check = 0xAAAAAAAA; //all odd bits is 1
151         return !((check & x)^ check);
152 }
153
```

Open ▾ bits.c ~/Desktop/Code/SS/hw1/datalab Save

```
154 //2
155 /*
156  * isLessOrEqual - if x <= y  then return 1, else return 0
157  *   Example: isLessOrEqual(4,5) = 1.
158  *   Legal ops: ! ~ & ^ | + << >>
159  *   Max ops: 24
160  *   Rating: 3
161  */
162 int isLessOrEqual(int x, int y)
163 {
164
165 //your codes here
166         int pomcheck = 1 << 31;
167         int pomx = (x & pomcheck) >> 31; //x is + or -?
168         int pomy = (y & pomcheck) >> 31; //y is + or -?
169         int ifsame = !(pomx ^ pomy); //x's symbol and y's symbol are the same?
170         int y_x = y + ~x + 1; //y-x
171         int pomy_x = y_x >> 31; //y-x is + or -?
172         return ((ifsame)&(!pomy_x))|(((!ifsame)&(pomx)); //same,check y-x;otherwise,- smaller
173 }
```

Open bits.c ~/Desktop/Code/SS/hw1/datalab Save

```
174
175 //3
176 /*
177  * logicalNeg - implement the ! operator, using all of
178  *              the legal operators except !
179  *   Examples: logicalNeg(3) = 0, logicalNeg(0) = 1
180  *   Legal ops: ~ & ^ | + << >>
181  *   Max ops: 12
182  *   Rating: 4
183  */
184 int logicalNeg(int x)
185 {
186 //your codes here
187         return ((x>>31)|((~x+1)>>31))+1; //0 ~0+1 symbol | is 0,others are all 1111...1111.
188 }
189
```

Open bits.c ~/Desktop/Code/SS/hw1/datalab Save

```
190 //4
191 //float
192 /*
193  * floatScale2 - Return bit-level equivalent of expression 2*f for
194  *   floating point argument f.
195  *   Both the argument and result are passed as unsigned int's, but
196  *   they are to be interpreted as the bit-level representation of
197  *   single-precision floating point values.
198  *   When argument is NaN, return argument
199  *   Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
200  *   Max ops: 30
201  *   Rating: 4
202  */
203
204 unsigned floatScale2(unsigned uf)
205 {
206 //your codes here
207         int exp = (uf & 0x7f800000) >> 23;
208         int sym = uf & (1 << 31);
209         if(exp == 0)
210         {
211                 return uf << 1 | sym;
212         }
213         if(exp == 255)
214         {
215                 return uf;
216         }
217         exp++;
218         if(exp == 255)
219         {
220                 return 0x7f800000 | sym;
221         }
222         return (exp << 23)|(uf & 0x807fffff);
223 }
```

Open bits.c ~/Desktop/Code/SS/hw1/datalab Save

```
225 //5
226 //float
227 /*
228  * floatFloat2Int - Return bit-level equivalent of expression (int) f
229  *   for floating point argument f.
230  *   Argument is passed as unsigned int, but
231  *   it is to be interpreted as the bit-level representation of a
232  *   single-precision floating point value.
233  *   Anything out of range (including NaN and infinity) should return
234  *   0x80000000u.
235  *   Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
236  *   Max ops: 30
237  *   Rating: 4
238  */
239 int floatFloat2Int(unsigned uf)
240 {
241 //your codes here
242         int sy = uf >> 31;
243         int exp = ((uf & 0x7f800000) >> 23) - 127;
244         int m = (uf & 0x007fffff) | 0x00800000;
```

```
245         if(!(uf & 0x7fffffff))
246         {
247                 return 0;
248         }
249         if(exp > 31)
250         {
251                 return 0x80000000;
252         }
253         if(exp < 0)
254         {
255                 return 0;
256         }
257         if(exp > 23)
258         {
259                 m <<= (exp - 23);
260         }
261         else
262         {
263                 m >>= (23 - exp);
264         }
265         if(!((m >> 31) ^ sy))
266         {
267                 return m;
268         }
269         else if(m >> 31)
270         {
271                 return 0x80000000;
272         }
273         else
274         {
275                 return ~m + 1;
276         }
277 }
```