北京大学-计算机系统导论
Data Lab：位操作

基于 GPT-4o 模型机翻得到，可能存在翻译错误/不准确之处，
请注意，课程鼓励大家阅读英文版原文以提高文档阅读水平，
此中文版仅为方便同学们阅读，避免遗漏重要事项，若有疑问请参阅英文版原文。

1 介绍

本次作业的目的是让你更加熟悉位级表示的常见模式、整数和浮点数。你将通过解决一系列编程“谜题”来实现这一目标。这些谜题中的许多都是刻意编写的，但在解决这些谜题的过程中，你会更多地思考位操作。

2 注意事项

- 这是一个个人项目。所有提交都通过 Autolab 服务进行电子提交。
- 你应该在自己的工作目录中使用 ICS Linux 服务器完成所有工作。

3 登录 Autolab

本学期所有的 ICS 实验都通过由 CMU 学生和教师开发的一个名为 *Autolab* 的 Web 服务提供。在你下载实验材料之前，你需要更新你的 Autolab 账户。在浏览器中输入 Autolab 主页

```
https://autolab.pku.edu.cn
```

你将被要求进行身份验证。首次验证后，Autolab 会提示你更新账户信息，包括一个**昵称**。昵称是 Autolab 为每个作业维护的公共记分板上识别你的外部名称，所以选择一个有趣的昵称吧！你可以随

时更改你的昵称。更新完账户信息后，点击“保存更改”按钮，然后选择“主页”链接继续进入 Autolab 的主页面。

你必须注册后才能获得 Autolab 账户。如果你是晚注册课程，可能不会被包含在 Autolab 的有效学生名单中。

4 讲义说明

你的实验材料包含在一个名为 `datalab-handout.tar` 的 Unix tar 文件中，你可以从 Autolab 下载。登录 Autolab 后

```
https://autolab.pku.edu.cn
```

你可以通过选择“Data Lab->Download handout”来获取 `datalab-handout.tar` 文件。首先将 `datalab-handout.tar` 复制到你计划工作的 Linux 工作目录中。然后输入命令

```
linux> tar xvf datalab-handout.tar
```

这将创建一个名为 `datalab-handout` 的目录，其中包含多个文件。你唯一需要修改和提交的文件是 `bits.c`。

`bits.c` 文件包含每个 16 个编程谜题的骨架。你的任务是根据严格的*编码规则*完成每个函数骨架：对于整数谜题，你只能使用*直线式*代码（即没有循环或条件）和有限数量的 C 算术和逻辑运算符。具体来说，你*仅*可以使用以下八个运算符：

```
! ~ & ^ | + << >>
```

有些函数进一步限制了这个列表。此外，不允许使用超过 8 位的常量。详见 `bits.c` 中的注释，了解每个函数的详细规则和编码规则讨论。

警告：不要让 Windows 的 WinZip 程序打开你的 `.tar` 文件（许多 Web 浏览器默认会自动执行此操作）。相反，将文件保存到你的 Linux 工作目录中，并使用 Linux 的 `tar` 程序解压文件。一般来说，在这门课中，绝对不要使用 Linux 以外的平台修改文件，否则可能导致数据（和重要工作）的丢失。

5 谜题

本节描述了你将在 `bits.c` 中解决的谜题。

5.1 位操作

表 1描述了一组操作和测试位集合的函数。“评分”字段给出了谜题的难度评分（得分），而“最大操作数”字段给出了实现每个函数允许使用的最大操作数。详见 `bits.c` 中的注释，了解函数的期望行为。你也可以参考 `tests.c` 中的测试函数。这些函数用作参考函数，以表达你的函数的正确行为，尽管它们不符合你的函数的编码规则。

名称	描述	评分	最大操作数
`bitAnd(x,y)`	使用仅~ 和| 实现x&y	1	8
`bitConditional(x,y,z)`	分别为每个位实现x?y:z	1	4
`implication(x,y)`	返回逻辑连接 →	2	5
`rotateRight(x,n)`	将 x 向右旋转 n 位	3	25
`bang(x)`	不使用! 计算!x	4	12
`countTrailingZero(x)`	计算 x 的二进制形式中的尾随零的数量	4	40

表 1: 位级操作函数。

5.2 二进制补码运算

表 2描述了一组使用二进制补码表示整数的函数。在这一部分，你将更好地理解整数在计算机系统中的表示方式。再次参考 `bits.c` 中的注释和 `tests.c` 中的参考版本，了解更多详细信息!

名称	描述	评分	最大操作数
`divpwr2(x,n)`	计算x/(2^n)，其中0<=n<=30	2	15
`sameSign(x,y)`	如果 x 和 y 符号相同则返回 1，否则返回 0	2	5
`multFiveEighths(x)`	将 x 乘以 5/8，向 0 取整	3	12
`satMul3(x)`	将 x 乘以 3，如果溢出则饱和到 T_{min} 或 T_{max}	3	25
`isLessOrEqual(x,y)`	如果x<=y 则返回 1，否则返回 0	3	24
`ilog2(x)`	计算 x 的以 2 为底的对数的底数，x>0	4	90

表 2: 算术函数

5.3 浮点运算

在这部分作业中，你将实现一些常见的浮点运算。在这一部分中，你可以使用标准的控制结构（条件语句、循环），并且可以使用 int 和 unsigned 数据类型，包括任意的无符号和整数常量。你不能使用任何联合、结构体或数组。最重要的是，你不能使用任何浮点数据类型、操作或常量。相反，任何浮点操作数都将作为 unsigned 类型传递给函数，任何返回的浮点值也将是 unsigned 类型。你的代码应该执行实现指定浮点运算的位操作。

表 3描述了一组操作浮点数位级表示的函数。参考 bits.c 中的注释和 tests.c 中的参考版本以获取更多信息。

名称	描述	评分	最大操作数
float_twice(x)	计算2*x	4	30
float_i2f(x)	将 int x 转换为 float	4	30
float64_f2i(x)	将 double x 转换为 int	4	20
float_negpwr2(x)	计算2^(-x)	4	20

表 3: 浮点函数。

自带的程序 fshow 帮助你理解浮点数的结构。要编译 fshow，切换到讲义目录并输入：

```
linux> make
```

你可以使用 fshow 查看任意模式表示的浮点数：

```
linux> ./fshow 2080374784

Floating point value 2.658455992e+36
Bit Representation 0x7c000000, sign = 0, exponent = f8, fraction = 000000
Normalized.  1.0000000000 X 2^(121)
```

你还可以给 fshow 提供十六进制和浮点值，它会解释它们的位结构。

```
linux> ./fshow 0x15213

Floating point value 1.212781782e-40
Bit Representation 0x00015213, sign = 0, exponent = 0x00, fraction = 0x015213
Denormalized.  +0.0103172064 X 2^(-126)
```

```
linux> ./fshow 15.213

Floating point value 15.2130003
Bit Representation 0x41736873, sign = 0, exponent = 0x82, fraction = 0x736873
Normalized.  +1.9016250372 X 2^(3)
```

6 评估

你的得分将基于以下分布从最高 80 分计算：

48 代码的正确性。

32 代码的性能，基于每个函数使用的操作数数量。

正确性分数。你必须解决的 16 个谜题的难度评分在 1 到 4 之间，其加权总和为 48。我们将使用`dlc`编译器检查你的函数是否遵循编码规则。我们将使用 BDD 检查器验证你的函数是否正确。只有当你的谜题符合所有编码规则并通过 BDD 检查器执行的所有测试时，你才能获得谜题的满分，否则将没有得分。

性能分数。目前我们主要关心你能否得到正确答案。然而，我们希望你能保持代码尽可能简短和简单。此外，有些谜题可以通过蛮力解决，但我们希望你更聪明。因此，对于每个函数，我们设定了一个最大操作数限制。这个限制非常宽松，旨在捕捉极其低效的解决方案。我们将使用`dlc`编译器验证你是否满足操作数限制。对于每个满足操作数限制的正确函数，你将获得两分。

7 自动评分你的工作

我们在讲义目录中包含了一些方便的自动评分工具——`btest`、`dlc`、BDD 检查器和 `driver.pl`——以帮助你检查工作的正确性。

- **`btest`:** 该程序通过多次调用 `bits.c` 中的函数并使用许多不同的参数值来检查函数的功能正确性。要构建和使用它，请输入以下两个命令：

  ```
  linux> make
  linux> ./btest
  ```

  或者你想要一句话完成：

```
linux> make && ./btest
```

注意，每次修改 bits.c 文件后，都必须重新构建 btest。

你会发现使用 btest 逐个函数工作会非常有帮助，逐个测试每个函数。你可以使用-f 标志指示 btest 仅测试单个函数：

```
linux> ./btest -f bitXnor
```

这将多次调用bitXnor函数，并使用许多不同的输入值。你可以使用-1、-2 和-3 选项标志向 btest 提供特定的函数参数：

```
linux> ./btest -f bitXnor -1 7 -2 0xf
```

这将只调用 bitXnor 一次，使用参数 x=7 和 y=15。如果你想通过插入 printf 语句来调试你的解决方案，请使用此功能；否则，你会得到过多的输出。

- **dlc:** 这是 MIT CILK 小组修改的 ANSI C 编译器，你可以用它来检查每个谜题的编码规则合规性。典型用法是：

```
linux> ./dlc bits.c
```

该程序在检测到问题（如非法运算符、操作数过多或整数谜题中的非直线代码）时会静默运行。使用-e 开关运行：

```
linux> ./dlc -e bits.c
```

会使 dlc 打印每个函数使用的操作数数量。输入./dlc -help 以获取命令行选项列表。

- **BDD 检查器:** 代码中的 btest 只是针对许多不同的输入值测试你的函数。对于大多数函数，可能的参数组合数量远超过可以全面测试的数量。为了提供全面覆盖，我们创建了一个称为**二进制决策图**（BDDs）的正式验证程序，称为 cbit，它可以全面测试你的函数的所有可能参数组合。

  你不直接调用 cbit。相反，有一系列 Perl 脚本设置和评估对它的调用。执行

```
linux> ./bddcheck/check.pl -f fun
```

以检查函数 fun。执行

```
linux> ./bddcheck/check.pl
```

以检查所有函数。执行

```
linux> ./bddcheck/check.pl -g
```

以检查所有函数并获得结果的紧凑表格摘要。

- **`driver.pl`:** 这是一个使用 `dlc` 和 BDD 检查器计算你的解决方案的正确性和性能分数的驱动程序。这是 Autolab 在自动评分你的提交时使用的程序。执行

```
linux> ./driver.pl
```

以检查所有函数并以紧凑表格格式显示结果。

8 提交说明

与过去可能参加的其他课程不同，在这门课程中，你可以在实验截止日期之前随时提交你的工作。

要获得分数，你需要使用 Autolab 选项“提交你的工作”上传你的 `bits.c` 文件。每次提交代码时，服务器都会在你的提交文件上运行驱动程序并生成评分报告（它还会将结果发布在记分板上）。服务器会存档每次提交和生成的评分报告，你可以随时使用“查看提交历史”选项查看。

提交注意事项:

- 在任何时间点，你最近上传的文件是你的正式提交。你可以随时提交。
- 每次提交时，你应使用“查看提交历史和分数”选项确认你的提交已正确自动评分。手动刷新页面以查看自动评分结果。
- 你必须在提交之前从 `bits.c` 文件中删除任何多余的打印语句。

9 建议

- 建议每次在使用 `make` 之前使用指令 `make clean`。
- 请参阅 `http://autolab.pku.edu.cn/faq.html` 以获取常见问题的解答。
- 如果遇到任何连接、基础 Linux 操作的问题，你可以前往 `https://ics.huh.moe/` 发帖询问。

- 你可以使用以下方式登录到课程的计算机上进行此项作业：

```
linux> ssh ubuntu@10.129.xx.xxx
```

- 一次测试和调试一个函数。我们建议如下步骤：
  - **步骤 1.** 使用 btest 测试和调试一个函数。首先，使用 -1 和 -2 参数结合 -f 调用一个函数，并传递一组特定的输入参数：

    ```
    linux> ./btest -f bitXnor -1 23 -2 0xabcd
    ```

    可以随意使用 printf 语句显示中间变量的值。但调试完函数后，请务必删除这些语句。
  - **步骤 2.** 使用 btest -f 检查函数在大量不同输入值下的正确性：

    ```
    linux> ./btest -f bitXnor
    ```

    如果 btest 检测到错误，它会打印出失败的具体输入参数。返回步骤 1，使用这些参数调试你的函数。
  - **步骤 3.** 使用 dlc 检查是否遵循了编码规则：

    ```
    linux> ./dlc bits.c
    ```
  - **步骤 4.** 在函数通过 btest 的所有测试后，使用 BDD 检查器进行最终的正确性测试：

    ```
    linux> ./bddcheck/check.pl -f bitXnor
    ```
  - **步骤 5.** 对每个函数重复步骤 1-4。在任何时候，你都可以运行 driver 程序来计算你获得的正确性和性能分数总数：

    ```
    linux> ./driver.pl
    ```

- 一些关于 dlc 的提示：
  - 不要在你的 bits.c 文件中包含 <stdio.h> 头文件，因为它会让 dlc 产生一些非直观的错误信息。虽然不包含 <stdio.h> 头文件，你仍然可以在调试时使用 printf，但 gcc 会打印一个可以忽略的警告。
  - dlc 程序执行的声明比 C++ 或 Java 甚至 gcc 更严格。特别是，任何声明必须出现在块（用大括号括起来的部分）中的任何非声明语句之前。例如，它会对以下代码提示错误：

    ```
    int foo(int x)
    {
      int a = x;
      a *= 3;      /* 这个语句不是声明 */
      int b = a;  /* 错误：此处不允许声明 */
    }
    ```

- 一些关于 BDD 检查器的提示：
  - BDD 检查器无法处理调用其他函数的函数，包括 `printf`。你应该使用 `btest` 来评估带有调试 `printf` 语句的代码。在提交代码之前，务必删除这些调试语句。
  - BDD 检查器脚本对代码的格式有些挑剔。他们期望函数以以下形式的行开头：

    ```
    int fun (...)
    ```

    或

    ```
    unsigned fun (...)
    ```

    并以左侧列唯一的右大括号结束。那应该是函数中唯一一个在左侧列的右大括号。

祝你好运！