作业二

本次作业一共有2道大题：

- 第1题是基础题，通过对比阅读源代码与汇编代码，巩固大家对课上介绍的汇编知识的掌握；
- 第2题是分析题，需要根据给定的情况进行分析，对课上所学进行补充。

Q1 基础

Q1.1 匹配（3分；1空1分）

```c
int choice1(int x) { return (x < 0); }

int choice2(int x) { return (x << 31) & 1; }

int choice3(int x) { return 15 * x; }

int choice4(int x) { return (x + 15) / 4; }

int choice5(int x) { return x / 16; }

int choice6(int x) { return (x >> 31); }
```

对于下列的汇编代码段，从上述C代码中找出其对应的函数实现（填写函数名即可）。

> 为方便讲解，我们以 `(旧)` 、 `(新)` 表示 `caller` 和 `callee` 下的寄存器值。

观察汇编代码可知这是X86-32的汇编代码，是通过运行栈来传递参数的。在 `caller` 栈帧中保存了返回地址（在 `[%esp]` 处）、若干参数（以 `[%esp + 4]` 为始的连续空间）；调用函数后进入 `callee` ，先保存当前 `%ebp` ，并将 `callee` 的 `%ebp(新)` 置为 `caller` 的 `%esp(旧)` 值，从而创建新的栈帧。

分配 `callee` 栈帧后，此时参数 `x` 所在的地址为 `%ebp(新) + 4 + 4 = %ebp(新) + 8` 。结束若干操作后，在返回前将返回值保存在 `%eax` 中，并还原 `%esp(旧)` 和 `%ebp(旧)` 。

汇编代码	对应函数 (填写函数名即可)	说明

`foo1:`
`pushl %ebp`
`movl %esp, %ebp`
`movl 8(%ebp), %eax`
`sall $4, %eax`
`subl 8(%ebp), %eax`
`movl %ebp, %esp`
`popl %ebp`
`ret`
	`choice3`	关键操作简化为 `(x << 4) - x` ，即 `x * (16 - 1) = x * 15`

汇编代码	对应函数 (填写函数名即可)	说明

`foo2:`
`  pushl  %ebp`
`  movl  %esp, %ebp`
`  movl  8(%ebp), %eax`
`  testl  %eax, %eax`
`  jge  .L2`
`  addl  $15, %eax`
`.L2:`
`  sarl  $4, %eax`
`  movl  %ebp, %esp`
`  popl  %ebp`
`  ret`
	`choice5`	通过`testl`指令设置`ZF`和`SF`（并清空`CF`,`OF`），
		若`SF`为`0`（表示正数），则直接跳转到`.L2`处执行`>> 4`，
		否则先`+ 15`再继续 这等价于向零舍入的带符号整数除法 **（见课件《2.1整数》P.28）**

`foo3:`
`  pushl  %ebp`
`  movl  %esp, %ebp`
`  movl  8(%ebp), %eax`
`  shrl  $31, %eax`
`  movl  %ebp, %esp`
`  popl  %ebp`
`  ret`
	`choice1`	关键的地方是逻辑右移指令`shrl`，在C语言中对于`x >> 4`，
		若`x`是`int`则进行算术右移 **（见 课件《2.1整数》P.27）**，
		若`x`是`unsigned`才进行逻辑右移：这里的参数是`int x`。
		因此最合适的是`choice1`返回符号位（最高位）的值
		（`0`或`1`）。

C语言中 >> 对应的汇编代码

带符号整数右移为算术右移

```
zyx@p100-2:/extradisk/zyx/intro2cs$ cat test.c
int foo3(int x) { return x >> 31; }

int main(int argc, char const *argv[]) {
    foo3(-123);
    return 0;
}
zyx@p100-2:/extradisk/zyx/intro2cs$ gcc -S -fno-stack-protector test.c -o test.S
zyx@p100-2:/extradisk/zyx/intro2cs$ cat test.S
        .file   "test.c"
        .text
        .globl  foo3
        .type   foo3, @function
foo3:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    %edi, -4(%rbp)
        movl    -4(%rbp), %eax
        sarl    $31, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   foo3, .-foo3
        .globl  main
        .type   main, @function
```

无符号整数右移为逻辑右移

```
zyx@p100-2:/extradisk/zyx/intro2cs$ cat test.c
int foo3(unsigned int x) { return x >> 31; }

int main(int argc, char const *argv[]) {
    foo3(-123);
    return 0;
}
zyx@p100-2:/extradisk/zyx/intro2cs$ gcc -S -fno-stack-protector test.c -o test.S
zyx@p100-2:/extradisk/zyx/intro2cs$ cat test.S
        .file   "test.c"
        .text
        .globl  foo3
        .type   foo3, @function
foo3:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    %edi, -4(%rbp)
        movl    -4(%rbp), %eax
        shrl    $31, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   foo3, .-foo3
        .globl  main
        .type   main, @function
```

Q1.2 `for`循环（4分；1空1分）

我们实现了如下的C语言函数：

```
int foo(int* a, int n, int val) {
    int i;
    for (i = n - 1; i >= 0 && val != a[i]; --i);
    return i;
}
```

根据上述函数，填写下面汇编代码中缺失的部分：

```
foo:
    leal        -1(__空格1__), %eax
.L2:
    testl       __空格2__, %eax
    js          .L1
    movslq      %eax, %rcx
    cmpl        %edx, (__空格3__,%rcx,4)
    __空格4__   .L1
    subl        $1, %eax
    jmp         .L2
.L1:
    ret
```

> FIXME： `cmpl` 指令的寄存器应为 `%edx` 而非 `%rdx` ，在参考解答中已修正。

观察可知变量 `i` 对应寄存器 `%rax` ，由此推断 `leal` 指令实际完成 `i = n - 1;` ，而参数 `n` 对应 `%rsi` 。

`testl` 指令则用于检查 `i >= 0` ：通过按位与判断符号位并设置 `SF` 。若 `SF=1` 则在 `js` 指令处发生跳转，直接结束。一般检查某个值是否为负数的表示为 `testl %eax, %eax` ，即自身"与"上自身；不过 `testl` 允许 Src operand 为立即数，因此 `testl $0x80000000, %eax` 也是可以的，但 Dst operand 不能是立即数，**因此 `testl %eax, $0x80000000` 是非法的**。

`cmpl` 指令则对应 `val != a[i]` ，考察的是 `a[i]` 的汇编表示：参数 `*a` 对应寄存器 `%rdi` ，而在上一条的 `movslq` 指令中， `%rcx = (符号扩展)(%eax)` ，又因 `sizeof(int) == 4` ，因此 `(%rdi, %rcx, 4)` 等价于 `a[i]` 。 `cmpl` 比较是 `val` 和 `a[i]` 的值，如果**等于**则立即返回，对应的下一条汇编为 `je` 。

空格	值
`__空格1__`	`%rsi`
`__空格2__`	`%eax` 或 `$0x80000000`
`__空格3__`	`%rdi`
`__空格4__`	`je`

Q1.3 `switch` 语句（5分，1 case 1分）

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $3, %rdi
    je  .L8
    jg  .L3
    cmpq    $1, %rdi
    je  .L4
    cmpq    $2, %rdi
    jne .L11
    movq    %rsi, %rax
    cqto
    idivq   %rcx
    addq    %rcx, %rax
    ret
.L3:
    movl    $1, %eax
    subq    $5, %rdi
    subq    %rdx, %rax
    cmpq    $2, %rdi
    movq    %rax, %rcx
    movl    $2, %eax
    cmovb   %rcx, %rax
    ret
.L8:
    movl    $1, %eax
    addq    %rcx, %rax
    ret
.L4:
    movq    %rdx, %rax
    imulq   %rsi, %rax
    ret
.L11:
    movl    $2, %eax
    ret
```

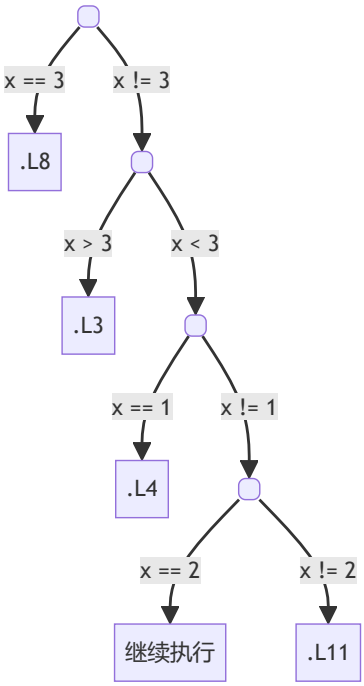
根据上述汇编代码，实现下方缺失的`switch`语句。

```
long switch_eg(long x, long y, long z) {
    long w = 1;
    switch (x) {
        // TODO: 实现switch
    }
    return w;
}
```

具体来说，你需要将`switch`语句中各分支的`case`和执行语句填写在下表中。题目中填写了`default`及对应的语句作为示例。注意：视情况需要包含`break;`；若无则填`-`。

变量`x`在`%rdi`，`y`在`%rsi`，`z`在`%rdx`，`w`作为返回值在`%rax`中。

从汇编代码可得到`switch`语句的实际分支跳转：



需要注意**在 `switch` 之前已经执行了 `int w = 1;`**。这时分情况讨论：

- `x == 3`：跳转到 `.L8`
  - 先执行 `w = 1;`，再执行 `w = w + z;` 后就返回了；
  - `w += z; break;`
- `x > 3`：跳转到 `.L3`
  - 先执行 `x -= 5;`（实际没有修改 `x` 的值），在赋值 `%rax` 前令 `%rcx = w - z`，`%rax = 2`；执行 `cmpq $2, %rdi`，来判断 `(x - 5) < 2` 是否为真（看 `(x - 5) - 2` 是否需要借位，若需要则置 `CF = 1`，反之为 `0`）；若成立（对应于 `CF = 1`）则执行 `cmovb` 指令，令 `%rax = %rcx`，否则保持 `%rax`；
  - 注意，在判断时不会考虑是否为有符号数：`x = 4` 时，`x - 5 = 0xffffffff > 2`，`x > 6` 时，`x - 5 >= 2`，它们对应的赋值为 `w = 2;`，**注意只有 `default` 分支有立即数赋值**，因此对应于 `default` 分支；`x - 5 < 2`，只有可能是 `x = 5` 和 `x = 6`，对应的赋值为 `w = w - z;`，执行完毕后返回；

    ```
    case 5:
    case 6:
        w -= z;
        break;
    // 其他情况与default一致
    ```

- `x == 1`：跳转到 `.L4`
  - 先执行 `w = z`，再执行 `w = w * y`，之后返回；
  - `w = z * y; break;`
- `x == 2`：继续执行
  - 先执行 `w = y`，再执行 `w = w / z`（对应于 `cqto` 和 `idivq %rcx` 指令），最后执行 `w = w + z` 后返回；
  - `w = y / z; w += z; break;`
  - 考虑到 `w += z; break;` 其实对应于 `x == 3`，可以按Fall through来写：

    ```
    case 2:
        w = y / z;
    case 3:
        w += z;
        break;
    ```

最终整合如下：

`case`	**执行语句（若需要应包含 `break;`；若无则填 `-`）**
`default`	`w = 2; break;`
`1`	`w = y * z; break;`
`2`	`w = y / z;`

case	执行语句（若需要应包含 break; ；若无则填 – ）
3	w += z; break;
5	–
6	w -= z; break;

对应的 switch 语句：

```
switch (x) {
    case 1:
        w = y * z; break;
    case 2:
        w = y / z;
    case 3:
        w += z; break;
    case 5:
    case 6:
        w -= z; break;
    default:
        w = 2; break;
}
```

另外一种（冗余但直观的）写法：

case	执行语句（若需要应包含 break; ；若无则填 – ）
default	w = 2; break;
1	w = y * z; break;
2	w = y / z; w += z; break;
3	w += z; break;
5	w -= z; break;
6	w -= z; break;

```
switch (x) {
    case 1:
        w = y * z; break;
    case 2:
        w = y / z; w += z; break;
    case 3:
        w += z; break;
    case 5:
        w -= z; break;
    case 6:
        w -= z; break;
    default:
        w = 2; break;
}
```

> **观察什么时候 switch 才会真的使用跳转表**
>
> 通过不停增加 含实际、独立的执行体的分支 数量，观察 switch 使用跳转表的时机。从中判断什么条件下 gcc 会倾向于将 switch 优化为分支，而不是创建跳转表来实现。

Q2 分析

Q2.1 整数 & 浮点数回顾（5分；1表达式1分，答案和解释各0.5分）

假设定义了以下变量：

```
int x, int y;
unsigned ux = (unsigned)(x);
unsigned uy = (unsigned)(y);

float f;
double d;
assert(!isnan(f) && !isnan(d)); // 保证 f 和 d 都不是 NaN
```

判断下面的表达式是否为真，若为假请解释或给出反例。

表达式	T/F	解释
`x == (int)(float)x`	F	`float`尾数为23位，不足以完整表示`int`，因此会出现精度损失
`(d + f) - d == f`	F	`(d + f)`按双精度运算，其结果的精度得到提升，最后会比`f`精确，因此二者并不相等
`if (x <= 0) then -x >= 0`	F	$T_{\min}$（即`INT_MAX = 0x80000000`）的负值还等于$T_{\min}$
`x >> 4 == x / 16`	F	移位运算不会使结果向0舍入
`(x|-x) >> 31 == -1`	F	反例，`x = 0`

Q2.2 条件跳转指令（3分；结果1分，解释2分）

X86-64指令集架构中的条件跳转指令`jg`是用于带符号数比较还是无符号数比较的？（1分）
其产生跳转的成立条件是`~(SF^OF)&~ZF`为真，请解释为何是这一条件。（2分）

有符号数比较；`jg`是 greater（signed）跳转，即`if(a - b) > 0`；

两种情况：

1. 当`OF == 0`时，则代表没有溢出，此时`SF`必须为`1`，`SF`为`1`代表结果为负，即`a - b < 0`，代表`a < b`；
2. 当`OF == 1`时，则代表有溢出，而此时`SF`必须为`0`，即结果最后为正数，那么此时则是负溢出，也可以得到`a - b < 0`，即`a < b`。

综上，`SF ^ OF`则代表小于的意思，`ZF`代表等于，所以`~(SF ^ OF) & ~ZF`代表大于。

Q2.3 分支跳转指示（5分；答对+解释 4分，优化1分）

在课上我们介绍了`likely()`和`unlikely()`指示宏，它们可以用于告诉编译器某个条件的成立概率，以便编译器进行优化。早期这两个宏是作为编译器专有扩展提供的，如今已被纳入C++20标准中，使用方式如下：

```cpp
// branch_pred.cpp
#include <cstdlib>

int main(int argc, char *argv[]) {
    int a, b;
    /* Get the value from somewhere GCC can't optimize */
    a = atoi(argv[1]); /*string to int*/
    b = a * a;
    if (a == 2) [[likely]] { // likely
        a++;
        b++;
    } else {
        a--;
        b--;
    }
    return a + b;
}
```

使用支持C++20标准的编译器（如`g++-11`）编译：

```
$ g++-11 -O2 branch_pred.cpp -o branch_pred
```

尝试执行该程序：

```
$ ./branch_pred 5
$ echo $?
28
```

> **`echo $?` 是什么意思?**
>
> 以往编程中，我们约定`return 0;`表示正常退出，非零值都是有问题的！这些值其实表示当前程序在执行结束后的 退出状态 (exit status)，称为退出码 (exit code)。
>
> 与HTTP状态码 (HTTP Status Code)（比如常说的404了）一样，通过观察退出码，用户可以判断程序的执行是否正常，以及产生的异常类型。在早期高级语言没有提供 异常机制 时，这能帮助大家锁定问题所在。通过`echo $?`可以查看上一次执行的退出码。
>
> 一个常见的使用场景是在环境配置脚本中，检查当前虚拟环境下Python的某个package是否导入成功，若不成功则进行安装：
>
> ```bash
> #!/bin/bash
> # ./exec_app.sh
>
> PYPI_MIRROR=https://pypi.tuna.tsinghua.edu.cn/simple
>
> # * 启用已存在的虚拟环境（若无则先创建）
> if [[ ! -d ".venv" ]]; then
>     python -m venv .venv
> fi
> source .venv/bin/activate
>
> # ^ 检查 numpy 是否成功导入，若否则重新安装
> python -c "import numpy as np"
> if [[ $? != 0 ]]; then
>     pip install --force-reinstall --no-cache-dir -i ${PYPI_MIRROR} numpy
> fi
> ```
>
> PS：如果你使用 Oh My Zsh （并搭配合适的主题，如PowerLevel10k）， `zsh` 会显示上一次执行的退出码，比 `bash` 直观。

观察`branch_pred.cpp`，我们最终通过`main()`的`return`语句将最终结果作为退出状态值返回，通过`echo $?`来查看结果。

对程序进行反汇编：

```
$ objdump -d test > test.asm
```

该命令会将反汇编的结果从标准输出流 `stdout` 重定向输出 到 `test.asm` 文件中保存。你可以使用编辑器（如VSCode）打开文件，或者用 `cat` 命令：

```
$ cat test.asm
```

试判断下面两段反汇编代码分别对应 `[[likely]]` 还是 `[[unlikely]]` （需要加上 `[[]]` ），并说明理由。（4分）

`[[likely]]` 和 `[[unlikely]]` 会把更可能执行的代码顺序排放在分支之后。

反汇编代码	对应的指示宏	理由
0000000000001060 <main>: 1060: f3 0f 1e fa endbr64 1064: 48 83 ec 08 sub $0x8,%rsp 1068: 48 8b 7e 08 mov 0x8(%rsi),%rdi 106c: ba 0a 00 00 00 mov $0xa,%edx 1071: 31 f6 xor %esi,%esi 1073: e8 d8 ff ff ff callq 1050 <strtol@plt> 1078: 89 c2 mov %eax,%edx 107a: 0f af d0 imul %eax,%edx 107d: 83 f8 02 cmp $0x2,%eax 1080: 74 0d je 108f <main+0x2f> 1082: 83 e8 01 sub $0x1,%eax 1085: 83 ea 01 sub $0x1,%edx 1088: 01 d0 add %edx,%eax 108a: 48 83 c4 08 add $0x8,%rsp 108e: c3 retq 108f: ba 05 00 00 00 mov $0x5,%edx 1094: b8 03 00 00 00 mov $0x3,%eax 1099: eb ed jmp 1088 <main+0x28> 109b: 0f 1f 44 00 00 nopl 0x0(%rax,%rax,1)	`[[unlikely]]`	可以发现 `0x1080` 处的指令判断 如果 `a == 2` ，那么 `jmp` 到 `0x108f` ，如果不相等，执行后续的减法操作。减法是更可能被执行的一个分支， 所以是 `[[unlikely]]`
0000000000001060 <main>: 1060: f3 0f 1e fa endbr64 1064: 48 83 ec 08 sub $0x8,%rsp 1068: 48 8b 7e 08 mov 0x8(%rsi),%rdi 106c: ba 0a 00 00 00 mov $0xa,%edx 1071: 31 f6 xor %esi,%esi 1073: e8 d8 ff ff ff callq 1050 <strtol@plt> 1078: 41 b8 08 00 00 00 mov $0x8,%r8d 107e: 83 f8 02 cmp $0x2,%eax 1081: 75 08 jne 108b <main+0x2b> 1083: 44 89 c0 mov %r8d,%eax 1086: 48 83 c4 08 add $0x8,%rsp 108a: c3 retq 108b: 89 c2 mov %eax,%edx 108d: 0f af d0 imul %eax,%edx 1090: 44 8d 44 02 fe lea -0x2(%rdx,%rax,1),%r8d 1095: eb ec jmp 1083 <main+0x23> 1097: 66 0f 1f 84 00 00 00 nopw 0x0(%rax,%rax,1) 109e: 00 00	`[[likely]]`	在分支 `0x1081` 处判断是否 `a !=2` ， 分支随后的指令将直接返回 `0x8` 也就是 `a == 2` 的结果， `a == 2` 是更可能发生的分支，所以是 `[[likely]]`

试说明 `[[likely]]` 和 `[[unlikely]]` 为什么在一定程度上能在优化程序在流水线处理器上的执行？（1分）

> 只要说明 "实际跳转会停止已进入流水线的指令的处理，造成性能损失，（默认大概率不会跳转时）因此尽量避免跳转" 即可得分。

以下解释为什么处理器执行时更倾向于 "默认预测分支不会发生跳转"，属于课外内容，供大家参考。

因为在存储层次结构中，靠近 CPU 的 Cache（后续课程会讲解，对应CSAPP 第）是以 cacheline 为单位（通常为 64 Bytes）搬运数据的，一个cacheline中包含多条指令，将更可能跳转的指令和分支指令排在一起，可以把该跳转方向的指令和分支指令一起搬运进Cache，从而提高 指令 Cache 的命中率。

另外现代 CPU 中集成了Next-Line等指令预取器，会将更靠近分支的指令提前预取进 Cache。