# 作业三

本次作业一共分为4道大题，**总分30分**：

- Q1 是综合题：考察对课上内容的掌握；
- Q2 是分析题：具体来说你需要分析汇编代码的执行过程及其用途；
- Q3 为实践题：通过实际的X86汇编编程，让大家能更好的理解X86汇编；
- Q4 为多选题：与课调委沟通后，我们希望收集大家对作业的反馈，一同改进课程质量。

> ⚠ **打包提交本次作业**
>
> 具体来说你需要将 书面作业的答案 + 经 `check.py` 测试 的 `fib.s` 两个文件**打包上传**到网络学堂。
>
> 假如你的学号为 `2022011451` ，则可以通过 `tar` 或 `zip` 命令打包：
>
> ```
> $ ls .  # 显示当前作业目录
> fib.s  作业三_答案.pdf/.docx/.md
> $ tar -czvf 2022011451.tar.gz *
> $ zip 2022011451.zip *
> ```
>
> 将打包好的 `2022011451.tar.gz` 或 `2022011451.zip` 上传到网络学堂。

# Q1. 综合（7分）

## Q1.1. 填充（5分）

编程解决猴子吃桃问题：每天吃一半再多吃一个，第十天想吃时候只剩一个，问总共有多少。
该程序的 C语言程序如下：

```
int eat_peaches(int i) {
    if (i == 10) return 1;
    else return (eat_peaches(i + 1) + 1) * 2;
}
```

填充汇编代码（Linux X86-64）中缺失的内容：

```
eat_peaches:
.LFB0:
    cmpl        __空格1__, %edi
    __空格2__   .L8
    movl        $1, %eax
    ret
.L8:
    __空格3__    $8, %rsp
    addl        $1, %edi
    call        eat_peaches
    leal        __空格4__(%rax,%rax), %eax
    addq        $8, __空格5__
    ret
```

本题每一空格1分。

`.LFB0` 部分判断 `i == 10` ，如果一致则执行 `if` 分支，直接返回 `1` ，否则跳转至 `.L8` 执行 `else` 分支；因此可以确定 `__空格(1)__` 为 `$10` ， `__空格(2)__` 为 `jne`

注意到 `eat_peaches` 其实是一个递归函数，因此在 `.L8` 中需要分配与回收栈帧，结合上下文可知 `__空格(3)__` 为 `subq` ， `__空格(5)__` 为 `%rsp`

实际的返回值为 `eat_peaches(i + 1)` 的返回值加 `1` 再乘 `2` ，即 `2 * eat_peaches(i + 1) + 2` ，在 `leal` 指令中写法应为 `2(%rax, %rax)` ，即 `%rax + %rax + 2` ，因此 `__空格(4)__` 为 `2`

| 空格 | 值 |
| --- | --- |
| `__空格(1)__` | `$10` |
| `__空格(2)__` | `jne` |
| `__空格(3)__` | `subq` |
| `__空格(4)__` | `2` |
| `__空格(5)__` | `%rsp` |

## Q1.2. `bfloat16` （2分）

`bfloat16` 是由Google提出的一种半精度浮点数， `exp` 域为8位， `frac` 域为7位， `sign` 域为1位。除了位宽度差别外， `bfloat16` 的其它规格符合IEEE 754标准。

现定义一个C语言 `union` （联合体）数据类型，如下所示：

```
union {
    bfloat16 f;
    unsigned short s;
}
```

现在为 `f` 赋予 `bfloat16` 所能表示的最接近于 `1` 且大于 `1` 的数，在X86（小端序）机器上运行时， `s` 的十六进制值为多少?

符合IEEE 754标准，即 `bfloat16` 严格遵循$(-1)^s \times M \times 2^E$的标准，且有$E = \exp - \text{bias} \Rightarrow \exp = E + \text{bias}$

> **为什么需要bias?**
>
> bias的用意在于将带符号数$E$的表示范围"平移"至无符号exp的表示范围中，从而避免产生歧义
>
> > 参见课件 《2.1 整数》第13页、《2.2 浮点数》第9页

接近于1且大于1，说明这个数：

- 是一个规格化浮点数 $\Rightarrow$ 尾数省略前导 `1` 、$\text{bias} = 2^{E-1} - 1$
- 正数，符号域$s = 0$，即 `s = 0b`
- **大于 `1`** ，说明$M = 1.0000001$，省略前导 `1` ，即 `frac = 0000001b`
- $E = 0$，则$\text{bias} = 2^{8-1} - 1 = 127$，于是 `exp = 0b + 01111111b = 01111111b`

则二进制表示为 `0 01111111 0000001b = 0x3f81`

s = 0x3F81

# Q2 Getter & Setter（10分）

过程调用以及返回的顺序在一般情况下都是"过程返回的顺序恰好与调用顺序相反"，但是我们可以利用汇编以及对运行栈的理解来编写汇编过程打破这一惯例。最典型的应用为有栈协程切换。

有如下汇编代码（**X86-32** 架构），其中 GET 过程唯一的输入参数是一个用于存储当前处理器以及栈信息的内存块地址（假设该内存块的空间足够大），而 SET 过程则用于恢复被 GET 过程所保存的处理器及栈信息，其唯一的输入参数也是该内存块地址。

```
GET:                                SET:
movl 4(%esp), %eax   #(A)           movl 4(%esp), %eax
...                                 ...
movl %edi, 20(%eax)                 movl 20(%eax), %edi
movl %esi, 24(%eax)                 movl 24(%eax), %esi
movl %ebp, 28(%eax)                 movl 28(%eax), %ebp
movl %ebx, 36(%eax)                 movl 36(%eax), %ebx
movl %edx, 40(%eax)                 movl 40(%eax), %edx
movl %ecx, 44(%eax)                 movl 44(%eax), %ecx
movl $1, 48(%eax)                   movl _____(%eax), %esp   #(D)
movl (%esp), %ecx    #(B)           pushl 60(%eax)           #(E)
movl %ecx, 60(%eax)                 movl 48(%eax), %eax
leal 4(%esp), %ecx   #(C)           ret
movl %ecx, 72(%eax)
movl 44(%eax), %ecx
movl $0, %eax
ret
```

在理解代码的基础上，回答下列问题：

1. SET 过程的返回地址是什么，其返回值是多少？（
2. 代码段中的 (A) 指令执行后， eax 中存放的是什么？
   (B) 指令执行后， ecx 中存放的是什么？
   (C) 指令的作用是什么？
   (E) 指令的作用是什么？
   并将 (D) 指令补充完整。

本题评分标准为 **基础分3分 + 正确性 7分**：每一小问1分，至少可得3分。

1. SET 过程的返回地址是 被 GET 过程保存的处理器执行的下一条指令的地址；返回值为 1
2. - (A) 指令执行后， eax 存放的是 保存处理器和栈信息的内存块的地址
   - (B) 指令执行后， ecx 存放的是 返回地址（即 调用 GET 过程后要执行的下一条指令所处的地址）
   - (C) 指令 记录 esp 栈顶指针，用于在 (D) 在将返回地址压入栈中
   - (D) 指令完整应为 `movl 72(%eax), %esp`
   - (E) 指令 将返回地址压入栈

# Q3 Fibonnaci求解（8分）

Fibonacci数列是一个非常经典的数列，其定义如下：

$$F(n) = F(n-1) + F(n-2)$$
$$F(0) = 0, F(1) = 1$$

为了巩固对X86-64汇编的掌握，我们提供了一个基于X86-64和Linux系统调用、 **求解Fibonacci数列中第$n$项** 的手写汇编代码，但是里面有一些bug，需要交给你来修复。 **(预期用时：< 8小时)**

详见 `fib.s` 的 `TODO:` 以及对应的 `hint` 。具体来说，你需要：

1. 手动实现 `print_int` 函数缺失的 `itoa_digits` （4分）

```
.type print_int, @function
print_int:
    # number to be printed in %rdi
    pushq %rbp
    movq  %rsp, %rbp
    movq %rdi, %rax

    # 先令换行符入栈，最后输出时换行
    movq $0xa, %rsi
    pushq %rsi
    movq $1, %rbx   # 记录当前待输出的字符数量，因为含'\n'所以初始为1

itoa_digits:
    # TODO: 将数字转换为字符串，并逆序保存在栈上，使得打印时次序正确
    # hint0: 从低位开始向高位处理，低位先入栈，高位先出栈
    # hint1: 使用cqto指令从32位扩展至64位，再用idivq指令获取 商（%rax）和 余数（%rdx）
    # hint2: 通过加上 $DIGIT_0 将余数转换为字符，并压入栈中保存，注意压入的是 X86-64寄存器
    # hint3: 更新待输出的字符总数（%rbx）中，这将作为print_digits的结束依据
```

   提示：可以先注释掉 `call fib` ，通过调用 `print_int` 函数，观察是否能正确输出 `scan_int` 的结果。

2. 修正 `fib.s` 中的 `fib` 函数（4分）

```
.type fib, @function
fib:
    # TODO: 修正fib函数，使得能够正确计算fibonacci数列
    # hint0: 可以使用GDB打断点，进行指令粒度的调试
    # hint1: 检查栈帧的分配与回收是否正确
    # hint2: 检查函数调用前后寄存器是否被正确保存与恢复
    # hint3: 检查函数调用时的参数是否正确传递

    # n is in %rdi
    pushq %rbp
    movq  %rsp, %rbp
    movq %rdi, %rax
    cmpq $2, %rdi
    jl fib_end
    decq %rdi
    call fib
    movq %rax, %r8
    decq %rdi
    call fib
    addq %r8, %rax
fib_end:
    movq %rsp, %rbp
    popq %rbp
    ret
```

   提示：你可以手写一个C程序，生成汇编代码后交叉对比，找出 `fib` 函数中的bug；善用 `pushq` / `popq` 指令能优化栈的使用体验。

使得 `fib.s` 最终能通过编译并正确执行。

本题给分方式为 **参与分 6分 + 正确分 2分**：即完成书面作业其它题目（不论正确性），却未能通过 `check.py` 随机化测试，仍可获得6分。

## Q3.1 `itoa_digits`

`print_int` 的本质其实就是实现 `itoa()` ，然后逐一遍历输出字符，对应的C++伪代码如下：

```
void print_int(int number) {
    std::stack<char> stk;
    stk.push('\n');

    int quotient, remainder; // 商、余数
    quotient = number;
    while (quotient) { // itoa_digits
        remainder = quotient % 10;
        stk.push((char)('0' + remainder));
        quotient /= 10;
    }

    while (!stk.empty()) { // print_digits
        char poll = stk.top();
        putchar(poll);
        stk.pop();
    }
}
```

我们需要做的就是将上面 `itoa_digits` 循环"翻译"为X86-64汇编代码。
首先要做的，就是搞清楚除法与取模操作分别对应什么汇编指令；按照作业提示与 `hint1` ，可以确定基本的指令为 `cqto` 和 `divq` （或 `idivq` ）

| 指令 | 效果（= 表示赋值） | 描述 |
| --- | --- | --- |
| `cqto` | `R[%rdx]:R[%rax] = 符号扩展(R[%rax])` | `%rax` 中的数符号扩展至128位，高64位保存在 `%rdx` 中，低64位保存在 `%rax` |
| `divq S` | `R[%rdx] = R[%rdx]:R[%rax] % S`<br>`R[%rax] = R[%rdx]:R[%rax] / S` | 无符号除法：除数为 `S` 中的值，被除数为经 `cqto` 扩展后128位，余数保存在 `%rdx` 中，商保存在 `%rax` 中 |
| `idivq S` | `R[%rdx] = R[%rdx]:R[%rax] % S`<br>`R[%rax] = R[%rdx]:R[%rax] / S` | 带符号除法，同上 |

> 由于$F(n) \geq 0$，因此这里使用 `divq` 或 `idivq` 都可以

**执行 `divq` 前，必须先执行 `cqto`** ，否则每次迭代前必须手动置 `%rdx` 为 `0` ，否则会进入死循环，引起SEGFAULT（可以通过 `gdb` 指令级调试）

获得余数后，我们通过（ `addq` 指令）加上 `'0'` （保存在常量 `$DIGIT_0` 中）将其转换为ASCII字符，并（调用 `pushq` 指令）压入运行栈中；重复上述过程，直到商为 `0` ，即 `R[%rdx] == R[%rax] == 0`

注意到在伪代码中，我们可以通过查询 `stk.size() != 0` （即 `!stk.empty()` ）来判断 `print_digits` 是否结束，可实际的运行栈并没有这样的接口，因此我们需要维护一个变量，记录需要打印的字符数量，结合上下文（的注释）可知变量保存在 `%rbx` 中

参考实现如下：

```
itoa_digits:
    # TODO: 将数字转换为字符串，并逆序保存在栈上，使得打印时次序正确
    # hint0: 从低位开始向高位处理，低位先入栈，高位先出栈
    # hint1: 使用cqto指令从32位扩展至64位，再用divq指令获取 商（%rax）和 余数（%rdx）
    # hint2: 通过加上 $DIGIT_0 将余数转换为字符，并压入栈中保存，注意压入的是 X86-64寄存器
    # hint3: 更新待输出的字符总数（%rbx）中，这将作为print_digits循环结束的依据
    movq $10, %rcx
    cqto
    divq %rcx
    # quotient is in %rax, remainder is in %rdx
    addq $DIGIT_0, %rdx
    pushq %rdx
    incq %rbx
    cmpq $0, %rax
    jnz itoa_digits
```

## Q3.2 `fib`

对于 `fib` 函数，不难发现其实就是（最朴素的）递归实现，很容易写出标准的C程序：

```c
int fib(int n) {
    if (n < 2) return n;
    return fib(n - 1) + fib(n - 2);
}
```

将它转为X86-64汇编代码：

```
fib:
    # n in %rdi
    pushq   %rbp                # (1)
    pushq   %rbx                # (3) 进入子节点后，保存父节点的n，并在返回时恢复
    subq    $8, %rsp            # (1) 分配栈帧
    movl    %edi, %ebx          # (2) 拷贝n到别处，递归调用时直接修改原本的n（在%rdi中）
    cmpl    $1, %edi
    jle .L3                     # if (n <= 1), 亦即 if (n < 2)
    leal    -1(%rdi), %edi      # 计算 fib(n - 1)
    call    fib
    movl    %eax, %ebp          # 临时保存 fib(n - 1) 的结果
    leal    -2(%rbx), %edi      # 计算 fib(n - 2)
    call    fib
    addl    %ebp, %eax          # 计算 fib(n - 1) + fib(n - 2)
.L1:
    addq    $8, %rsp            # (4) 回收栈帧
    popq    %rbx                # (5) 恢复父节点的n
    popq    %rbp                # (4)
    ret
.L3:
    movl    %edi, %eax
    jmp .L1
```

需要注意的几处细节：

- 在每个过程实例中，寄存器的值是会变化的，因此需要把递归前后会复用的寄存器值保存到一个 仅属于当前实例的运行空间上，即当前实例的**栈帧**上
- 需要将 `%rdi` 保存到别处，这里是保存到 `%rbx` 寄存器上，并且在进入 子递归实例 后，保存在运行栈上，待返回前恢复（思考一下二叉树的非递归后序遍历是如何保存变量的）
- 递归调用 `fib(n - 1)` 和 `fib(n - 2)` 后，计算结果必须保存到运行栈中
- 判断何时及如何高效地执行 `if` 分支

比对生成的汇编代码，找出原本的 `fib` 实现中出现的问题（共有4处错误）：

```
.type fib, @function
fib:
    # TODO: 修正fib函数，使得能够正确计算fibonacci数列
    # hint0: 可以使用GDB打断点，进行指令粒度的调试
    # hint1: 检查栈帧的分配与回收是否正确
    # hint2: 检查函数调用前后寄存器是否被正确保存与恢复
    # hint3: 检查函数调用时的参数是否正确传递

    # n is in %rdi
    pushq %rbp
    movq  %rsp, %rbp    # 分配栈帧
    movq %rdi, %rax     # 对应于 if (n < 2)，直接返回 n
    cmpq $2, %rdi
    jl fib_end
    decq %rdi           # FIXME: 递归后%rdi不再被恢复了，需要提前保存在栈上
    call fib
    movq %rax, %r8
    decq %rdi           # FIXME: 如果%rdi保存在栈上但没恢复，那么这里的%rdi是某个后代节点的值
    call fib
    addq %r8, %rax      # FIXME: 递归后子节点及其后代很有可能修改了%r8，那么这个%r8是谁的？
fib_end:
    movq %rsp, %rbp     # FIXME: 回收栈帧
    popq %rbp
    ret
```

可以看见，这里 `fib` 最大的问题在于寄存器的保存与恢复；一种 ~~纯手写、极不优雅的~~ 实现方式如下：

```
.type fib, @function
fib:
    # TODO: 修正fib函数，使得能够正确计算fibonacci数列
    # hint0: 可以使用GDB打断点，进行指令粒度的调试
    # hint1: 检查栈帧的分配与回收是否正确
    # hint2: 检查函数调用前后寄存器是否被正确保存与恢复
    # hint3: 检查函数调用时的参数是否正确传递

    # n is in %rdi
    pushq %rbp
    movq  %rsp, %rbp
    movq %rdi, %rax
    cmpq $2, %rdi
    jl fib_end
    pushq %rdi
    decq %rdi
    call fib
    movq %rax, %r8
    popq %rdi
    pushq %r8
    subq $2, %rdi
    call fib
    popq %r8
    addq %r8, %rax
fib_end:
    movq %rbp, %rsp
    popq %rbp
    ret
```