# Homework6

这是 Homework6.pdf的标准答案。

注:“过程与解析”是 Gemini 给出的解析，并不是答案本身的一部分，仅供参考。

## Answer1

**函数原型：** `int fun(short c, char d, int *p, int x);`

**参数类型和顺序：**

- `c`: `short`
- `d`: `char`
- `p`: `int *`
- `x`: `int`

**过程与解析：**

IA32 体系结构中，函数参数从右向左入栈。在函数 prologue (`pushl %ebp`, `movl %esp, %ebp`) 之后，可以通过 `(%ebp)` 访问栈上的内容。

- `8(%ebp)`: 第 1 个参数 (c)
- `12(%ebp)`: 第 2 个参数 (d)
- `16(%ebp)`: 第 3 个参数 (p)
- `20(%ebp)`: 第 4 个参数 (x)

我们逐行分析汇编代码:

1. `Movsbl 12(%ebp), %edx`
    - `12(%ebp)` 是第 2 个参数 `d`。
    - `Movsbl` (Move Sign-extended Byte to Long) 指令将一个 1 字节(byte)的值符号扩展为 4 字节(long)并存入 `%edx`。
    - **推论：** `d` 的类型是 `char` (或 `signed char`)。
2. `Movl 16(%ebp), %eax`
    - `16(%ebp)` 是第 3 个参数 `p`。
    - `Movl` (Move Long) 指令将一个 4 字节的值存入 `%eax`。
    - **推论：** `p` 是一个 4 字节类型，可能是 `int`, `long` 或指针。
3. `Movl %edx, (%eax)`
    - `(%eax)` 表示将 `%eax` 中的值作为地址进行解引用（访问内存）。
    - 这条指令将 `%edx` (即 `d` 的值) 存入 `%eax` (即 `p` 的值) 所指向的内存地址。
    - 这对应 C 代码 `*p = d;`。
    - **推论：** `p` 必须是一个指针，即 `int *` (或 `long *`)。
4. `Movswl 8(%ebp), %eax`

- `8(%ebp)` 是第 1 个参数 `c`。
- `Movswl` (Move Sign-extended Word to Long) 指令将一个 2 字节(word)的值符号扩展为 4 字节(long)并存入 `%eax`。
- **推论：** `c` 的类型是 `short` (或 `signed short`)。

5. `Movl 20(%ebp), %edx`

   - `20(%ebp)` 是第 4 个参数 `x`。
   - `Movl` 指令将一个 4 字节的值存入 `%edx`。
   - **推论：** `x` 的类型是 `int` (或 `long`)。

6. `Subl %eax, %edx`

   - 执行 `%edx = %edx - %eax`。
   - 这对应 C 代码 `x - c` (因为 `x` 在 `%edx` 中，`c` 在 `%eax` 中)。

7. `Movl %edx, %eax`

   - 将 `Subl` 的结果 (即 `x - c`) 存入 `%eax` 寄存器。
   - `%eax` 是 IA32 中用于存放函数返回值的标准寄存器。
   - 这对应 C 代码 `return x - c;`。
   - **推论：** 函数的返回类型是 `int` (或 `long`)。

综上所述，参数从右到左入栈的顺序是 `x`, `p`, `d`, `c`。C 语言函数原型中的参数顺序与入栈顺序相反，因此原型为：`int fun(short c, char d, int *p, int x);`

---

# Answer2

| | %esp | %ebp |
|---|---|---|
| 1 | 0x7FFFFFC0 | 0x7FFFFFF4 |
| 2 | 0x7FFFFFC0 | 0x7FFFFFC0 |
| 3 | 0x7FFFFFC4 | 0x7FFFFFF4 |

**过程与解析：**

**初始状态：**

- `%esp = 0x7FFFFFC4`
- `%ebp = 0x7FFFFFF4`
- `Mem[0x7FFFFFC0] = 0x120`

**(1) Instruction 1: `pushl %ebp`**

- **过程：**
  1. `%esp` 减 4 (long) 字节：`0x7FFFFFC4 - 4 = 0x7FFFFFC0`。
  2. 将 `%ebp` 的值 (`0x7FFFFFF4`) 存入新的 `%esp` (`0x7FFFFFC0`) 指向的内存地址。`Mem[0x7FFFFFC0]` 被更新为 `0x7FFFFFF4`。
- **结果：**

- `%esp = 0x7FFFFFC0`
- `%ebp = 0x7FFFFFF4` (不变)

**(2) Instruction 2: `movl %esp, %ebp`**

- **过程：**
  1. 将 `%esp` 的值 (`0x7FFFFFC0`) 复制到 `%ebp`。
- **结果：**
  - `%esp = 0x7FFFFFC0` (不变)
  - `%ebp = 0x7FFFFFC0`

**(3) Instruction 3: `popl %ebp`**

- **过程：**
  1. 从 `%esp` (`0x7FFFFFC0`) 指向的内存地址读取值。`Mem[0x7FFFFFC0]` 的值是 `0x7FFFFFF4` (在步骤 1 中存入)。
  2. 将读取到的值 (`0x7FFFFFF4`) 存入 `%ebp`。
  3. `%esp` 加 4 字节：`0x7FFFFFC0 + 4 = 0x7FFFFFC4`。
- **结果：**
  - `%esp = 0x7FFFFFC4`
  - `%ebp = 0x7FFFFFF4`

Answer3(1)

**对应的 C 语言代码：**

```c
#include <stdio.h>

int main() {
    int var_b; // 对应 8(%rsp)
    int var_a; // 对应 12(%rsp)

    // 汇编分析:
    // leaq 8(%rsp), %rdx  -> arg3 = &var_b
    // leaq 12(%rsp), %rsi -> arg2 = &var_a
    // movl $.LC0, %edi    -> arg1 = "%d %d"
    scanf("%d %d", &var_a, &var_b);

    // movl 12(%rsp), %ecx -> %ecx = var_a
    // movl 8(%rsp), %edx  -> %edx = var_b
    // movl %edx, %esi     -> %esi = var_b
    // xorl %ecx, %esi     -> %esi = var_b ^ var_a
    // movl $.LC1, %edi    -> arg1 = "%d %d %d\n"

    // printf 调用的参数 (x86-64):
    // RDI: %edi (arg1) = "%d %d %d\n"
    // RSI: %esi (arg2) = var_b ^ var_a
    // RDX: %edx (arg3) = var_b
    // RCX: %ecx (arg4) = var_a
    printf("%d %d %d\n", var_b ^ var_a, var_b, var_a);
```

```
    return 0;
}
```

Answer3(2)

**Line 24 (`call printf`) 执行前的状态：**

(假设 `scanf` 读入的值为 `a_val` 和 `b_val`，分别存入 `12(%rsp)` 和 `8(%rsp)`)

**寄存器值：**

- `%rsp`: `0x8000400`
- `%edi`: `[Address of .LC1]` (即 `"%d %d %d\n"` 字符串的地址)
- `%esi`: `b_val ^ a_val` (变量 b 和 a 的异或结果)
- `%edx`: `b_val` (变量 b 的值)
- `%ecx`: `a_val` (变量 a 的值)

**栈状态图 (高地址在顶部)：**

| 地址 | 内容 | 说明 |
|---|---|---|
| 0x8000420 | | <-- 进入 main 前的 %rsp |
| 0x8000418 | [ main 的返回地址 (8 B) ] | |
| 0x8000410 | [ (栈帧中未使用的 8 B) ] | |
| 0x800040C | [ 局部变量 a (4 B, a_val) ] | <-- 12(%rsp) |
| 0x8000408 | [ 局部变量 b (4 B, b_val) ] | <-- 8(%rsp) |
| 0x8000400 | [ (栈帧中未使用的 8 B) ] | <-- %rsp (当前栈顶) |

**过程与解析：**

1. **进入 `main` 函数：**
    - `call main` 指令执行 (由 C 运行时库发起)。
    - `%rsp` (初始值 `0x8000420`) 减 8，用于存放 64 位的返回地址。
    - `%rsp` 变为 `0x8000418`。
    - `Mem[0x8000418]` 存有返回地址。
2. **`subq $24, %rsp`：**
    - 为 `main` 函数分配 24 字节的栈帧。
    - `%rsp = 0x8000418 - 24 = 0x8000400`。
    - 栈帧范围是 `0x8000400` 到 `0x8000417`。
3. **局部变量定位 (`scanf`)：**
    - `leaq 12(%rsp), %rsi`: 计算 `0x8000400 + 12 = 0x800040C`。这是变量 a 的地址。
    - `leaq 8(%rsp), %rdx`: 计算 `0x8000400 + 8 = 0x8000408`。这是变量 b 的地址。

- `call __isoc99_scanf`：`scanf` 将读取的值存入 `Mem[0x800040C]` (a_val) 和 `Mem[0x8000408]` (b_val)。

4. **准备 `printf` 参数 (Line 24 之前)：**
   - `movl 12(%rsp), %ecx`：`%ecx = Mem[0x800040C] = a_val`。
   - `movl 8(%rsp), %edx`：`%edx = Mem[0x8000408] = b_val`。
   - `movl %edx, %esi`：`%esi = b_val`。
   - `xorl %ecx, %esi`：`%esi = %esi ^ %ecx = b_val ^ a_val`。
   - `movl $.LC1, %edi`：`%edi` 指向格式化字符串 `.LC1`。
   - 此时，`%rsp` 仍然是 `0x8000400`。