

# Homework5

这是 [Homework5.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

Place	Value
%eax	0x10000000
%ecx	22
\$0x10000004	0x10000004
0x10000012	327680 (或 0x00050000)
0xFFFFFFFF8	NONE
(%eax,%ecx,8)	44

### 过程与解析:

- 1. **\$0x10000004**: 这是一个立即数。
- 2. **0x10000012**: 这是一个内存地址。
  - 基地址为 **0x10000000**。偏移量为**0x12** (十进制 18) 字节。
  - 这是一个未对齐的地址。我们需要查看它跨越的 `int` 值。
  - $\&a[0][4] = 0x10000000 + (0*10 + 4) * 4 = 0x100000010$ 。
  - $\&a[0][5] = 0x10000000 + (0*10 + 5) * 4 = 0x100000014$ 。
  - `a[0][4]` 的值为  $0*10 + 4 = 4$  (**0x00000004**)。
  - `a[0][5]` 的值为  $0*10 + 5 = 5$  (**0x00000005**)。
  - 内存布局 (小端序):
    - ... 10 11 12 13 14 15 16 17 ... (地址)
    - ... 04 00 00 00 05 00 00 00 ... (字节)
  - 从 **0x10000012** 开始读取 4 字节，读到的是 **00 00 05 00**。
  - 小端序解释为 **0x00050000**，即十进制 **327680**。
- 3. **0xFFFFFFFF8**: 这通常是一个栈地址 (例如 `EBP` 的负偏移量)。
  - 函数 `f()` 有局部变量 `i` 和 `j`，它们存储在 `f()` 的栈帧上。
  - 当 `f()` 返回后，其栈帧被销毁。该地址的内存内容是未定义的。
- 4. **(%eax,%ecx,8)**: 这是一个内存地址计算，使用表中给定的值。
  - 地址 = `%eax + (%ecx * 8)`
  - 地址 = **0x10000000 + (22 \* 8)**
  - 地址 = **0x10000000 + 176 (0xB0) = 0x100000B0**。
  - 我们需要找到这个地址的值。偏移量为 176 字节。
  - $176 / \text{sizeof(int)} = 176 / 4 = 44$ 。
  - 这指向数组的第 44 个元素 (0-indexed)。
  - $k = i * \text{LEN} + j \Rightarrow 44 = i * 10 + j$ 。解得  $i = 4, j = 4$ 。
  - 该地址是 `&a[4][4]`。

- 函数 `f()` 执行  $a[4][4] = 4 * 10 + 4 = 44$ 。

## Answer2

```
int dw_loop(int x, int y, int n)
{
    do
    {
        /* C code for the body */
        x = x + n;
        y = y * n;
        n = n - 1;
    } while (/* C code for the condition */
            n > 0 && y < n
            );
    return x;
}
```

### 过程与解析:

#### 1. 寄存器分配:

- `%eax = x`
- `%ecx = y`
- `%edx = n`

#### 2. `.L2`: (循环体):

- `addl %edx, %eax`:  $eax = eax + edx \rightarrow x = x + n$ ;
- `imull %edx, %ecx`:  $ecx = ecx * edx \rightarrow y = y * n$ ;
- `subl $1, %edx`:  $edx = edx - 1 \rightarrow n = n - 1$ ;

#### 3. 循环条件判断:

- `testl %edx, %edx`: 检查  $n$ 。
- `jle .L5`: 如果  $n \leq 0$ , 跳出循环。
- `cmpl %edx, %ecx`: 比较  $y$  和  $n$  ( $y - n$ )。
- `j1 .L2`: 如果  $y < n$  (less), 跳回 `.L2` 继续循环。

4. **C 逻辑**: 循环继续的条件是  $(n > 0)$  并且  $(y < n)$ 。这就是 `while` 的条件。`testl/jle` 是  $n > 0$  的高效实现（通过检查  $n \leq 0$  来退出），这是 `&&` 运算符的短路求值。

## Answer3(1)

```
; C: int cmov_complex(int x, int y)
; x at 8(%ebp), y at 12(%ebp)
cmov_complex:
    pushl %ebp
    movl %esp, %ebp

    movl 8(%ebp), %eax      ; eax = x
    movl 12(%ebp), %ecx     ; ecx = y

    ; 计算 res2 = (x + y) * y (x >= y 的情况)
```

```

movl %eax, %edx      ; edx = x
addl %ecx, %edx      ; edx = x + y
imull %ecx, %edx     ; edx = (x + y) * y [res2]

; 计算 res1 = x * y (x < y 的情况)
imull %ecx, %eax     ; eax = x * y [res1]

; 比较
cmpl %ecx, 8(%ebp)   ; 比较 x 和 y (从内存中重新读取 x)

; 条件传送
cmovl %eax, %edx     ; if (x < y) [Less], edx = eax (res1)
                    ; 否则, edx 保持为 res2

movl %edx, %eax      ; 将最终结果放入 eax

popl %ebp
ret

```

### 过程与解析:

1. **cmov** 的思想是先计算两个分支的结果，然后根据条件选择一个。
2.  $res1 = x * y$
3.  $res2 = (x + y) * y$
4. 我们分别将  $res1$  计算到 **%eax**，将  $res2$  计算到 **%edx**。
5. **cmpl %ecx, 8(%ebp)** 比较  $x$  和  $y$ 。
6. **cmovl %eax, %edx** 表示 "如果 ( $x < y$ )，则将 **%eax** ( $res1$ ) 的值传送到 **%edx**"。
7. 如果条件不成立 ( $x \geq y$ )，**%edx** 保持其原始值 ( $res2$ )。
8. 最终结果在 **%edx** 中，将其移至 **%eax** 作为返回值。

## Answer3(2)

### 解释:

**gcc** 不使用 **cmov** 是因为在这个特定情况下，使用 **cmov** 的开销比使用条件跳转 (**jge**) 更大。

1. **cmov 的代价:** **cmov** 指令本身很快，但它要求两个分支的计算都必须提前完成。在此代码中，两个分支都包含一个\*\***imull** (乘法)\*\* 操作，这是一个高延迟（昂贵）的指令。
  - **cmov** 路径 =  $(x+y) + (x*y) + ((x+y)*y) + \text{cmp} + \text{cmov}$
  - 这总是需要执行 **两个 imull** 操作。
2. **条件跳转的代价:** 使用条件跳转，CPU 只需要计算一个分支。
  - **jmp** 路径 =  $\text{cmp} + \text{jge} + (x+y) + (1 * \text{imull})$  (如果  $x \geq y$ )
  - **jmp** 路径 =  $\text{cmp} + \text{jge} + (1 * \text{imull})$  (如果  $x < y$ )
  - 这总是只执行 **一个 imull** 操作。
3. **权衡:** 编译器 **gcc** 评估了这两种策略。它认为，执行两个昂贵的 **imull** 操作的代价，高于执行一个 **imull** 操作并承担一次潜在的分支预测错误 (pipeline stall) 的代价。因此，它选择了更快的条件跳转方案。

(**cmov** 通常只在分支中的计算都非常简单时 (如 **add, mov**) 才更优。)

## Answer4

```

; 假设 x 在 %ecx, result 在 %edx
movl $0, %edx          ; result = 0
movl %ecx, %eax         ; eax = x
subl $24, %eax          ; eax = i = x - 24 (归一化索引)

; 检查索引是否在 [0, 6] 范围内
cmpl $6, %eax
ja .L_default          ; if (i > 6, unsigned), goto default

; 间接跳转
jmp *.L_table(,%eax,4)  ; 跳转到 L_table[i]

; --- 数据段 (只读) ---
.section .rodata
.align 4
.L_table:               ; 跳转表 (7个条目, 对应 x = 24 到 30)
    .long .L_24         ; i=0 (x=24)
    .long .L_default    ; i=1 (x=25, 缺失, 走 default)
    .long .L_26         ; i=2 (x=26)
    .long .L_27_28      ; i=3 (x=27)
    .long .L_27_28      ; i=4 (x=28)
    .long .L_29_30      ; i=5 (x=29)
    .long .L_29_30      ; i=6 (x=30)

; --- 代码段 ---
.section .text

.L_24:                  ; case 24
    movl %ecx, %edx     ; result = x
    addl %ecx, %edx     ; result = x + x
    jmp .L_end          ; break

.L_27_28:               ; case 27, 28
    movl %ecx, %edx     ; result = x
    addl $10, %edx      ; result = x + 10
    jmp .L_end          ; break

.L_26:                  ; case 26
    movl %ecx, %edx     ; result = x
    sall $1, %edx       ; result = x * 2
                        ; (无 break, fall-through)

.L_29_30:               ; case 29, 30
    addl $5, %edx       ; result = result + 5
    jmp .L_end          ; break

.L_default:             ; default case
    movl $3, %edx       ; result = 3
                        ; (隐式 fall-through 到 .L_end)

```

```
.L_end:  
; ... result 在 %edx 中 ...
```