

本次作业要求如下：

1.截止日期：2025/05/14 周日晚24:00

2.提交作业给本课程QQ群里的助教（庄养浩：QQ1473889198）

3.命名格式：附件和邮件命名统一为“第一次作业+学号+姓名”，作业为 PDF 格式；

4.注意：

(1) 选择、判断和填空只需要写答案，大题要求有详细过程，过程算分。

(2) 答案请用另一种颜色的笔回答，便于批改，否则视为无效答案。

(3) 大题的过程最好在纸上写完拍照放word里，或者直接在word里写。

(4) 本次作业由Part1、Part2、Part3，需要全部作答

5.本次作业遇到问题请联系课程群里的助教。

## Part1 信息的表示和处理+程序优化

### 一. 选择题

1. 位于存储器层次结构中的最顶部的是(A)。

A. 寄存器 B. 主存 C. 磁盘 D. 高速缓存

2. 关于 Intel 的现代 X86-64 CPU，说法正确的是(B)。

A. 属于RISC B. 属于CISC C. 属于MISC D. 属于AVX

3. 操作系统在管理硬件时使用了几个抽象概念，其中(A)是对处理器、主存和 I/O 设备的抽象表示。

A. 进程 B. 虚拟存储器 C. 文件 D. 虚拟机

4. 以下有关编程语言的叙述中，正确的是(D)。

A. 计算机能直接执行高级语言程序和汇编语言程序

B. 机器语言可以通过汇编过程变成汇编语言

C. 汇编语言比高级语言有更好的可读性

D. 汇编语言和机器语言都与计算机系统结构相关

5. 给定字长的整数  $x$  和  $y$  按补码相加，和为  $s$ ，则发生正溢出的情况是(A)

A.  $x > 0, y > 0, s \leq 0$  B.  $x > 0, y < 0, s \leq 0$

C.  $x > 0, y < 0, s \geq 0$  D.  $x < 0, y < 0, s \geq 0$

6. 设机器数字长8位（含1位符号位），若机器数DAH为补码，分别对其进行算术左移一位和算术右移一位，其结果分别为 (A)

A. B4H, EDH

B. B5H, 6DH

C. B4H, 6DH

D. B5H, EDH

7. -1029的16位补码用十六进制表示为(C)。

A. 8405H

B. 0405H

C. FBFBH

D. 7BFBH

8. 已知两个正浮点数， $N_1 = 2^{j_1} \times S_1$ ， $N_2 = 2^{j_2} \times S_2$ ，当下列 (A) 成立时， $N_1 < N_2$ 。

A.  $S_1$ 和 $S_2$ 均为规格化数，且 $j_1 < j_2$

B.  $S_1 < S_2$

C.  $S_1$ 和 $S_2$ 均为规格化数，且 $j_1 > j_2$

D.  $j_1 < j_2$

9. C程序执行到整数或浮点变量除以 0 可能发生( D )。
- 显示除法溢出错直接退出
  - 程序不提示任何错误
  - 可由用户程序确定处理办法
  - 以上都可能
10. 补码加法运算的溢出判别中，以下说法正确的是( D )
- 符号相同的两个数相加必定不会发生溢出
  - 符号不同的两个数相加可能发生溢出
  - 符号相同的两个数相加必定发生溢出
  - 符号不同的两个数相加不可能发生溢出
11. 假定变量i、f的数据类型分别是int、float。已知i=12345，f=1.2345e3，则在一个32位机器中执行下列表达式时，结果为“假”的是( C )。
- $i == (int)(float)i$
  - $i == (int)(double)i$
  - $f == (float)(int)f$
  - $f == (float)(double)f$
12. 以下关系表达式，结果为“真”的是( B )。
- $2147483647U > -2147483648$
  - $(unsigned) -1 > -2$
  - $-1 < 0U$
  - $2147483647 < (int) 2147483648U$
13. 假定某数采用IEEE 754单精度浮点数格式表示为00000001H，则该数的值是( B )。
- NaN (非数)
  - $1.0 \times 2^{(-149)}$
  - $1.00...01 \times 2^{(-127)}$
  - $1.0 \times 2^{(-150)}$
14. C语言程序如下，下列说法叙述正确的是(D )。
- ```
#include <stdio.h>
#define DELTA sizeof(int)
int main(){
    int i;
    for (i = 40; i - DELTA >= 0; i -= DELTA)
        printf("%d ",i);
}
```
- 程序有编译错误
  - 程序输出10个数: 40 36 32 28 24 20 16 12 8 4 0
  - 程序死循环，不停地输出数值
  - 以上都不对
15. 若int型变量x的最高有效字节全变0，其余各位不变，则对应C语言表达式为( A )。
- $((unsigned) x \ll 8) \gg 8$
  - $((unsigned) x \gg 8) \ll 8$
  - $(x \ll 8) \gg 8$
  - $(x \gg 8) \ll 8$

## 二. 填空题

- 64 位系统中 short 数 -2 的机器数二进制表示为 111111111111110B。
- 判断整型变量n的位7为1的C语言表达式是 if(n>>7 & 1)。
- 1024采用IEEE 754单精度浮点数格式按内存地址从低到高表示的结果（十六进制表示，小端模式）是 0000084CH。
- C语言中的 double 类型浮点数用 64 位表示。
- 64 位系统中，整型变量x = -7，其在内存从低地址到高地址依次存放的数是

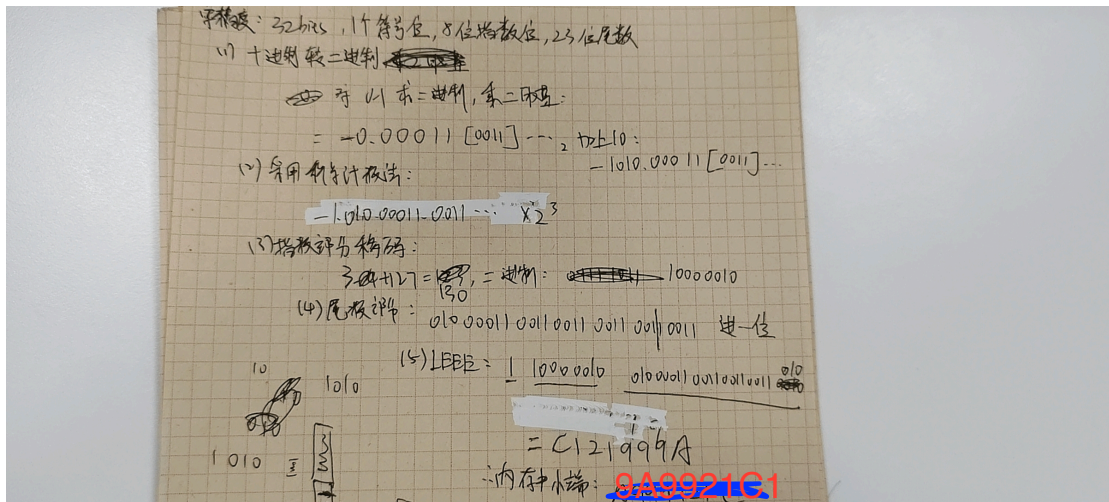
\_\_\_\_\_F9FFFFFFH\_\_\_\_\_ (十六进制表示, 小端模式)。

### 三. 判断题

1. (F) C浮点常数 IEEE754 编码的缺省舍入规则是四舍五入。
2. (T) 浮点数 IEEE754 标准中, 规格化数比非规格化数多。
3. (T) 对 unsigned int x,  $(x*x) \geq 0$  总成立。
4. (F) CPU 无法判断加法运算的和是否溢出。
5. (T) C语言中的有符号数强制转换成无符号数时位模式不会改变。
6. (F) C语言中数值从 int 转换成 double 后, 数值虽然不会溢出, 但是可能是不精确的。
7. (F) C语言中从 double 转换成 float 时, 值可能溢出, 但不可能被舍入。

### 四. 分析题

1. 请说明 float 类型编码格式, 并按步骤计算 -10.1 的各部分内容, 写出 -10.1 在内存从低地址到高地址的存储字节内容 (小端系统)。



2. 向量元素和计算的相关程序如下, 请改写或重写计算函数 `vector_sum`, 进行速度优化, 并简要说明优化的依据。(如果能自己动手在电脑上测试一下, 优化前后性能提升了多少会有额外加分, 贴上截图, 注明机器型号)

根据题目要求, 编写了一个测试程序

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h> // 添加时间测量头文件

/* 向量的数据结构定义 */
typedef struct {
    int len;
    float *data; // 向量元素的存储地址
} vec;

/* 获取向量长度 */
int vec_length(vec *v) { return v->len; } // 使用内联函数优化

/* 获取向量中指定下标的元素值，保存在指针参数val中 */
int get_vec_element(vec *v, int idx, float *val) {
    if (idx >= v->len) return 0;
    *val = v->data[idx];
    return 1;
}

/* 优化版本：直接访问数据 */
void vector_sum_optimized(vec *v, float *sum) {
    // todo
}

/* 原始版本 */
void vector_sum_original(vec *v, float *sum) {
    long int i;
    *sum = 0;
    for (i = 0; i < vec_length(v); i++) {
        float val;
        get_vec_element(v, i, &val);
        *sum += val;
    }
}
```

```

int main() {
    // 初始化测试数据（增大数据量便于观察时间差）
    const int size = 10000000; // 1000万元素
    float *a = malloc(size * sizeof(float));
    for (int i = 0; i < size; ++i) {
        a[i] = (i % 100) * 0.1f; // 生成测试数据
    }

    // 创建向量对象
    vec *temp = malloc(sizeof(vec));
    temp->data = a;
    temp->len = size;

    // 测试优化版本
    clock_t start = clock();
    float sum = 0;
    vector_sum_optimized(temp, &sum);
    clock_t end = clock();
    double opt_time = (double)(end - start) / CLOCKS_PER_SEC;

    // 测试原始版本
    start = clock();
    vector_sum_original(temp, &sum);
    end = clock();
    double orig_time = (double)(end - start) / CLOCKS_PER_SEC;

    // 输出结果
    printf("优化版本耗时: %.6f 秒\n", opt_time);
    printf("原始版本耗时: %.6f 秒\n", orig_time);
    printf("性能提升: %.2f 倍\n", orig_time / opt_time);

    // 释放资源
    free(a);
    free(temp);
    return 0;
}

```

观察原始求和函数，我们不难发现有如下局限性：

1. 函数调用：每次判断边界条件的时候都会调用一次vec\_length函数,可以拿到外面单独计算
2. 减少过程调用，每次关于get\_vec\_element的调用的开销都比较大，可以使用get\_vec\_start函数，只调用一次获取，然后每次通过数组访问

```
float *get_vec_start(vec *v)
{
    return v->data;
}
```

3. 消除不必要的内存引用：使用一个临时变量来累加每次取到的向量元素值
4. 循环展开：利用kX1循环展开提升性能

最终优化代码如下：

```
void vector_sum_optimized(vec *v, float *sum) {
    //todo
    long int i;
    *sum = 0;
    int x=vec_length(v)-2; //代码移动，防止每次都调用
    float *data=get_vec_start(v); //减少函数调用
    float t_sum=0; //临时变量，减少内存引用
    for (i = 0; i < x; i+=3) {
        t_sum = t_sum+data[i]+data[i+1]+data[i+2]; //3x1循环展开
    }
    for(;i<x+2;i++)
    {
        t_sum+=data[i]; //末尾数据处理
    }
    *sum = t_sum;
}
```

优化效果：

```
● fang@ubuntu-fang:~/fangtest$ gcc home_1.c
● fang@ubuntu-fang:~/fangtest$ ./a.out
  优化版本耗时: 0.100443 秒
  原始版本耗时: 0.228269 秒
  性能提升: 2.27 倍
● fang@ubuntu-fang:~/fangtest$ ./a.out
  优化版本耗时: 0.119145 秒
  原始版本耗时: 0.237177 秒
  性能提升: 1.99 倍
● fang@ubuntu-fang:~/fangtest$ ./a.out
  优化版本耗时: 0.108237 秒
  原始版本耗时: 0.246727 秒
  性能提升: 2.28 倍
● fang@ubuntu-fang:~/fangtest$ ./a.out
  优化版本耗时: 0.141508 秒
  原始版本耗时: 0.243684 秒
  性能提升: 1.72 倍
● fang@ubuntu-fang:~/fangtest$ ./a.out
  优化版本耗时: 0.100814 秒
  原始版本耗时: 0.263182 秒
  性能提升: 2.61 倍
○ fang@ubuntu-fang:~/fangtest$
```



```

/*向量的数据结构定义 */
typedef struct{
    int len;    //向量长度，即元素的个数
    float *data; //向量元素的存储地址
} vec;

/*获取向量长度*/
int vec_length(vec *v){return v->len;}

/* 获取向量中指定下标的元素值，保存在指针参数 val 中*/
int get_vec_element(*vec v, size_t idx, float *val){
    if (idx >= v->len)
        return 0;
    *val = v->data[idx];
    return 1;
}

/*计算向量元素的和*/
void vector_sum(vec *v, float *sum){
    long int i;
    *sum = 0; //初始化为 0
    for (i = 0; i < vec_length(v); i++) {
        float val;
        get_vec_element(v, i, &val); //获取向量 v 中第 i 个元素的值，存入 val 中
        *sum = *sum + val;    //将 val 累加到 sum 中
    }
}

```

## Part2 X64汇编

1. C语言程序中的整数常量、整数常量表达式是在（ A ）阶段初始化和计算的。  
(A) 预处理 (B) 执行 (C) 链接 (D) 编译
2. 关于Intel 的现代X86-64 CPU正确的是（ C ）  
A. 属于RISC B. 属于MISC C. 属于CISC D. 属于NISC
3. 下列叙述正确的是（ D ）  
A. X86-64指令"mov \$-1, %eax"会使%rax的值变成0xfffffffffffffffff  
B. 在一条指令执行期间，CPU不会两次访问内存  
C. CPU不总是执行CS::RIP所指向的指令，例如遇到call、ret指令时  
D. 一条mov指令不可以使用两个内存地址操作数
4. 在 x86-64 系统中，调用函数 int gt (long x, long y)时，保存参数 y 的寄存器是（ B ）  
A. %rdi B. %rsi C. %rax D. %rdx
5. 在 x86-64 系统中，调用函数 long gt (long x, long y)时，保存返回值的寄存器是（ C ）  
A. %rdi B. %rsi C. %rax D. %rbx
6. 下列传送指令中，哪一条是正确的（ D ）  
A. movb \$0x105, (%ebx)



- B. `movl %rdi,%rax`  
 C. `movq %rdi,$105`  
 D. `movl 4(%rsp),%eax`  
 7. C语言程序定义了结构体`struct noname{char c; long n; int k; float *p; short a;};`若该程序编译成64位可执行程序，则`sizeof(noname)`的值是32B。  
 8. 在X86-64中，程序的栈存放在内存中，栈顶元素的地址是所有栈中元素中最小的，栈指针寄存器`%rsp`保存着栈顶元素的地址。  
 9. While 循环的两种展开方法分别是什么：（ 跳到中间 ）（ guarded-do ）  
 下面代码是哪种？（ 跳到中间 ）

```
long fact_while_jm_goto(long n)
{
    long result = 1;
    goto test;
loop:
    result *= n;
    n = n-1;
test:
    if (n > 1)
        goto loop;
    return result;
}
```

10. 已知内存和寄存器中的数值情况如下：

| 内存地址  | 值    | 寄存器               | 值     |
|-------|------|-------------------|-------|
| 0x100 | 0xff | <code>%rax</code> | 0x100 |
| 0x104 | 0xAB | <code>%rcx</code> | 0x1   |
| 0x108 | 0x13 | <code>%rdx</code> | 0x3   |
| 0x10c | 0x11 |                   |       |

请填写下表，给出对应操作数的值：

| 操作数                        | 值     |
|----------------------------|-------|
| <code>%rax</code>          | 0x100 |
| <code>(%rax)</code>        | 0xff  |
| <code>9(%rax,%rdx)</code>  | 0x11  |
| <code>0xfc(,%rcx,4)</code> | 0xff  |
| <code>(%rax,%rdx,4)</code> | 0x11  |

11. 有下列C函数：

```
long max(long x, long y)
{
    long result;
    if(x >= y){
        result = x;
    }else{
```

```

        result = y;
    }
    return result;
}

```

请写出红色部分代码使用条件数据传输来实现条件分支的等价形式，并给出对应的条件传送指令(初始执行指令 `movq %rdi, %rax`)。

```

movq %rdi, %rax
cmpq %rsi, %rax
jae .L1
movq %rsi, %rax
.L1:
ret

```

12. 假设变量sp和dp被声明为类型:

`Src_t *sp;`

`Dest_t *dp;`

这里的Src\_t和Dest\_t是用typedef声明的数据类型，我们想使用适当的数据传送指令来实现下面的操作:

`*dp = (Dest_t) *sp;`

sp和dp的值分别存储在%rdi和%rsi中，对下表的每个表项，请写出合适的两条传送指令（如需用到其他寄存器，使用%rax）。注意：规定如果强制类型转换既涉及大小变化又涉及符号变化时，操作应先改变大小。

| Src_t         | Dest_t        | 指令                                                       |
|---------------|---------------|----------------------------------------------------------|
| char          | int           | <u>_①movsbl (%rdi), %eax</u><br><u>Movl %eax, (%rsi)</u> |
| char          | unsigned      | <u>_②movsbl (%rdi), %eax</u><br><u>Movl %eax, (%rsi)</u> |
| unsigned char | long          | <u>_③movzbq (%rdi), %rax</u><br><u>Movq %rax, (%rsi)</u> |
| int           | char          | <u>_④movl (%rdi), %eax</u><br><u>Movb %al, (%rsi)</u>    |
| unsigned      | unsigned char | <u>_⑤ movl (%rdi), %eax</u><br><u>Movb %al, (%rsi)</u>   |

13. 简述缓冲区溢出攻击的原理以及防范方法（2种）

**原理：**当程序向固定长度的缓冲区（如数组）写入数据时，若未检查输入数据的长度，攻击者可输入超长数据覆盖相邻内存区域，例如：1.覆盖函数返回地址，劫持程序执行流程（如跳转到攻击代码）。2,.修改关键变量或函数指针，引发非预期行为。

防范方法：1. 栈保护：在函数栈帧的返回地址前插入一个随机值（Canary），函数返回前验证该值是否被篡改。若检测到修改，立即终止程序。

2. 将内存页标记为不可执行（No-eXecute），即使攻击者注入恶意代码，CPU也会拒绝执行该区域指令。

### Part3 Y86处理器体系结构

- Y86-64的指令ret编码长度为（ A ）。  
A. 1字节    B. 2字节    C. 9字节    D. 10字节
- Y86-64的CPU顺序结构设计与实现中，分成（ B ）个阶段  
A. 5    B. 6    C. 7    D. 8
- Y86-64的CPU流水线结构设计与实现中，分成（ A ）个阶段  
A. 5    B. 6    C. 7    D. 8

判断题：

- Y86-64的顺序结构实现中，寄存器文件读时是作为时序逻辑器件看待（ F ）
- 现代超标量 CPU 指令的平均周期接近于 1 个但大于 1 个时钟周期（ F ）
- 下表是CPU的某个场景，解释：加载指令（mrmovq和popq）占有所有执行指令的20%，其中15%会导致 加载/使用 冒险。条件分支指令占有所有执行指令的25%，其中40%不选择分支。返回指令占有所有执行指令的3%。完成下表：

| 原因   | 名称 | 指令频率 | 条件频率 | 气泡数  | 总处罚  | CPI  |
|------|----|------|------|------|------|------|
| 加载使用 | lp | 0.20 | 0.15 | 0.03 | 0.32 | 1.32 |
| 预测错误 | mp | 0.25 | 0.40 | 0.2  |      |      |
| 返回   | rp | 0.03 | 1.00 | 0.09 |      |      |

- 在Y86-64架构的机器上，有下列汇编代码，请指出jne t指令之后执行的那条指令的地址为 0x00B （顺序执行，不考虑流水线）。

```
0x000:    xorq %rax, %rax
0x002:    jne  t
...
0x019: t:  irmovq $3, %rdx
0x023:    irmovq $4, %rcx
0x02d:    irmovq $5, %rdx
```

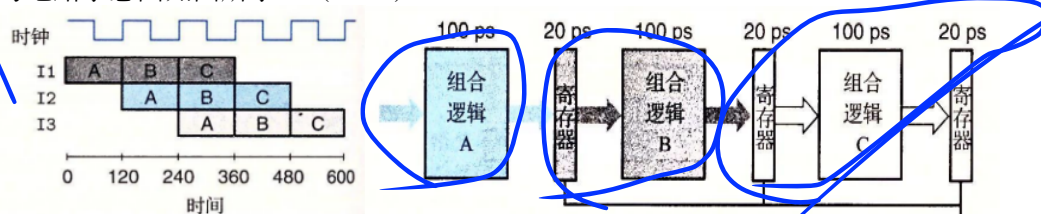
8. 请写出Y86-64的CPU流水线结构设计与实现中各流水线阶段的名称（注意顺序不要错位）。

取指  
译码  
执行  
访存  
写回

9. Y86-64流水线CPU中的冒险的种类与处理方法。

1. 数据冒险：指令使用寄存器R为目的，之后又使用R寄存器为源，处理方法有：暂停：在执行阶段插入气泡，使得当前指令暂停在译码阶段。数据转发：增加valM/valE的旁路路径，直接送到译码阶段
2. 加载使用冒险，指令暂停在取指和译码阶段，在执行阶段插入气泡，同时valM旁路转发
3. 控制冒险：分支预测错误，在条件为真的地址处的两条指令为bubble，ret：在ret后插入三个bubble

10. 假设有一个理想的三阶段流水线，执行指令为 I1 (Instruction1), I2, I3, 其时序图与电路示意图如图所示：（P321）



请分析不同时间点各寄存器中保存值所属指令，并完成下表（填入 I1, I2, I3, None。寄存器1 表示左数第一个寄存器）

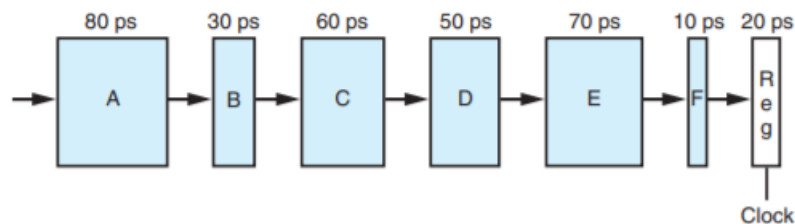
|     | 寄存器1 | 寄存器2 | 寄存器3 |
|-----|------|------|------|
| 239 | I1   | None | None |
| 241 | I2   | I1   | None |
| 300 | I2   | I1   | None |
| 361 | I3   | I2   | I1   |

11. 请根据描述在Y86-64顺序实现的条件下完成下面表格。

- (1) 请写出Y86-64顺序实现的push rA指令在各阶段的微操作。
- (2) 请写出Y86-64顺序实现的pop rA指令在各阶段的微操作。

| 指令   | push rA             | pop rA                       |
|------|---------------------|------------------------------|
| 取指   | Icode: ifunc<-M[PC] | Icode: ifunc<-M[PC]          |
|      | Ra:rb<=M[PC+1]      | Ra:rb<-M[PC+1]               |
|      | valP<-PC+2          | valP<-PC+2                   |
| 译码   | valA<-R[rA]         | valA<-R[%rsp]                |
|      | valB<-R[%rsp]       | valB<-R[%rsp]                |
| 执行   | valE<-valB-8        | valE<-valB+8                 |
| 访存   | M[valE]<-valA       | valM<-M[valA]                |
| 写回   | R[%rsp]<-val[E]     | R[ra]<-valM<br>R[%rsp]<-valE |
| 更新PC | PC<-valP            | PC<-valP                     |

12. 假设我们分析图中的组合逻辑，认为它可以分为6个块，以此命名为A、B、C、D、E、F，延迟分别为80，30，60，50，70，10（单位ps），如下图所示。



在这些块之间插入流水线寄存器，就得到这一设计的流水线化的版本。根据在哪里插入流水线寄存器，会出现不同的流水线深度(有多少个阶段)和最大吞吐量的组合。假设每个流水线寄存器的延迟为20ps。

请根据以上描述，回答下列问题：

(1) 只插入一个寄存器，得到一个两阶段的流水线。要使吞吐量最大化，该在哪里插入寄存器呢？吞吐量和延迟是多少？

在CD之间插入

吞吐量5.3GIPS

延迟：340ps

(2) 要使一个三阶段的流水线的吞吐量最大化，该将两个寄存器插在哪里呢？吞吐量和延迟是多少？

在BC, DE之间各插入一个寄存器

吞吐量：7.7GIPS

延迟：360ps

(3) 要使一个四阶段的流水线的吞吐量最大化，该将三个寄存器插在哪里呢？吞吐量和延迟是多少？

AB, CD, DE之间各插入一个  
吞吐量: 9.1GIPS  
延迟: 380ps

(4) 要得到一个吞吐量最大的设计，至少要有几个阶段？描述这个设计及其吞吐量和延迟。

至少5个阶段，只有EF阶段合并在一起，在每个组合逻辑后面都插入一个寄存器  
吞吐量: 10GIPS  
延迟: 400ps

至少，这样不改变最终吞吐量

13. 下面是一个程序段，请根据Y86-64的微指令和流水线数据相关的知识，试解释为什么在call指令之前要插入3个nop指令。

```
0x000:    irmovq Stack,%rsp    # Intialize stack pointer
0x00a:    nop
0x00b:    nop
0x00c:    nop
0x00d:    call p                # Procedure call
0x016:    irmovq $5,%rsi       # Return point
0x020:    halt
```

Irmovq的结果在写回阶段才能写回%rsp，但是call指令译码阶段需要取%rsp的数据，所以为了避免更改错误的值，需要插入3个nop，等上一条指令的写回阶段执行完再执行call的译码阶段