南开大学
Nankai University

南　　开　　大　　学

计　算　机　学　院

计算机组成原理第二、三章自出习题

冯思程

年级：2021 级

专业：计算机科学与技术

指导教师：张金

2023 年 4 月 30 日

目录

一、 习题

(一) 题目

题目代码

```
#include<iostream>

using namespace std;

int num=0;
float numsub=0.2;
int intarr[3]={-1,2,3};
float floatarr[3]={1.2,2.5,3.0};

int func(int a,int b){
int c=0;
c=a+b;
return c;
}
int main(){
while(num<4){
if(intarr[2]=num){
  intarr[2]+=num;
}
num++;
}

int a=1;
int b=2;
int c=func(a,b);
return 0;
}
```


这是一段小冯同学写的代码，下面他将提出一些习题给大家：

1.MIPS 指令集

- 1）小冯同学遇到了难题，他想知道代码第 16-21 行的 while 循环如何可以翻译成 MIPS 指令集中的具体指令，可以帮帮他吗，在关键代码处如果可以给出注释将更好不过了。
- 2）简单拿出上述指令中的一个说明其格式，并说明其字段结构。
- 3）脑筋急转弯：我们都知道在 MIPS 指令集设计过程中，所有指令长度都是一致的，但是有多种不同格式的指令，这体现了什么?
- 4）小冯同学还想知道在计算机内部，数组应该被如何存储，请以代码中的 intarr 数组为例进行简单说明。

- 5）小冯同学还对代码中的 func 函数调用过程比较感兴趣，那么你能帮助小冯同学理解在这个过程中，程序的控制权是如何转移到 func 中，又是如何从 func 函数中返回到主函数的吗?
- 6）快问快答：说明 c++ 程序在转换可执行文件并执行所需要的 4 个步骤（学过汇编课程的同学应该深有体会）
- 7）上面代码中，使用了栈和堆来辅助代码运行，其中栈是我们非常熟悉的结构，那么小冯同学还对其中的堆有一些生疏，但是他听说堆是一个“危险的果实”，请你帮忙对堆的使用风险进行说明。
- 8）基础问答：小冯同学想考你有关 MIPS 寻址模式的知识，请你选一种你熟悉的寻址模式进行说明。

2. 算术运算

- 1）在上面的代码中，多次运用到了加法，小冯同学查阅了相关资料发现使用多个全加器的效率很低，很影响性能，有什么改进的策略你可以给小冯同学说明一下?
- 2）快问快答：如何将减法看作加法一样运算，简单说明即可。
- 3）请提供一种方法：能够在某些情况下忽略溢出的发生，而在另一些情况下则能进行溢出的检测。
- 4）除了加减法，我们还需要乘除法来进行运算，普通的乘法器和除法器大家一定都很了解了，小冯同学想了解一下更加快速的乘法和除法，请你说明其硬件结构，并说明为什么他们的速度更快。
- 5）上面代码中，小冯同学声明了 float 类型的变量，但是没有进行运算是因为他不了解浮点运算的具体过程。请你以浮点数的乘法为例，具体说明进行乘法的过程。
- 6）在计算机的运算世界中，由于对性能的高要求，串行计算已经不能满足要求，需要对运算进行并行化，小冯同学发现计算机组成与设计书中介绍了有关子字并行的知识，请你简单介绍一下其原理和思路。针对上面的程序其中 while 循环是一个有可能可以进行并行化处理的地方，那么它能用 SIMD 进行并行化处理吗，说明原因。

(二) 覆盖知识点

1. 计算机指令集和汇编语言
2. MIPS 指令格式和字段结构
3. MIPS 指令集硬件设计原则
4. 计算机内存中数组的存储方式
5. C++ 函数调用过程和程序控制权的转移
6. 将 C++ 程序转换为可执行文件的步骤
7. 堆的使用风险以及温习栈的知识
8. MIPS 寻址模式

9. 加法器的优化以及原理
10. 减法运算转换为加法运算-减法与加法的同源本质
11. 忽略溢出和检测溢出的方法
12. 快速乘法和除法器的硬件结构和速度优势以及原理
13. 浮点数乘法的具体过程
14. 子字并行的原理和思路，以及对 while 循环的 SIMD 并行化处理的可行性。

(三) 设计思路

我的设计思路是：
首先发现第二、三章的知识点非常琐碎和繁琐，所以用一道小题进行整体概括是不太可能的，所以我决定进行一个情景连续题，逐步按照我们章节知识的深入进行提问，同时，为了增加题目的趣味性，我为题目设置了一个情景，小冯同学写代码的同时提出问题。然后具体题目，我参照教材和上课所学习的知识结合进行题目的编写。以上就是我全部的设计思路。

(四) 答案与解析

1.MIPS 指令集

- 1）下面是 while 循环部分的 MIPS 指令代码：

```
loop:    addi $8,$0,4              # 初始化循环计数器，$8 存储计数器值
         addi $9,$0,3              # 将 3 存储到$9 中
         bge $8,$9,exit            # 如果$8 >= $9，则跳转到 exit
         sll $10,$8,2              # $10 = $8 << 2，即$10 = $8 * 4
         add $11,$4,$10            # $11 = $4 + $10，即$11 = &intarr + $8 * 4
         lw $12,0($11)             # $12 = *$11，即取出 intarr[$8]的值
         beq $12,$8,add_num        # 如果$12 == $8，则跳转到 add_num
         addi $8,$8,1              # $8 = $8 + 1
         j loop                    # 跳转回 loop
add_num:add $12,$12,$8             # intarr[2] += num
         addi $8,$8,1              # $8 = $8 + 1
         j loop                    # 跳转回 loop
exit:    # while 循环结束
```

- 2）用上面代码 loop 后面的第一行代码进行示例：
  该指令为 I 型指令，其格式为：opcode rs rt immediate
  其中，opcode 为操作码字段，用于识别指令类型，addi 指令的 opcode 为 0x08；rs 为源寄存器字段，表示操作数的来源，其中数字 0 表示寄存器 0；rt 为目的寄存器字段，表示操作结果的存储位置，其中数字 8 表示寄存器 8；immediate 为立即数字段，表示需要加到源寄存器中的值，4 为该立即数字段的值，最多可表示 16 位的有符号整数。
- 3）体现了硬件设计的三大基本原则之一：优秀的设计需要适宜的折中方案。
- 4）在计算机内部，数组通常被存储在内存中的连续地址空间中，每个元素占据相邻的内存单元。对于 intarr 数组，由于是 int 类型，每个元素占据 4 个字节（32 位），因此 intarr[0] 会存储在某个内存地址处，而 intarr[1] 会存储在相邻的内存地址处，以此类推。

- 5）在 func 函数被调用时，程序的控制权会从主函数转移到 func 函数中，此时主函数的状态被保存到堆栈中。在 func 函数执行完成后，程序会从堆栈中恢复主函数的状态，同时返回值也被传递回主函数。这个过程通过调用指令和返回指令来实现。具体来说，当程序调用 func 函数时，会执行一个 jal 指令（跳转并链接）将程序计数器（PC）设置为 func 函数的地址，并将返回地址存储到寄存器 ra 中。在 func 函数执行完成后，会执行一个 jr 指令（跳转寄存器）将程序计数器设置为返回地址，从而返回到主函数。

- 6）将 C++ 程序转换为可执行文件并执行需要以下四个步骤：
  预处理：在此步骤中，预处理器会扫描源代码文件并展开所有的预处理指令。预处理器还将生成一个预处理文件，其中包含扩展后的源代码。

  编译：在此步骤中，编译器将预处理后的源代码转换为汇编代码。汇编代码是一种低级的表示形式，用于描述程序的控制流、数据存储和处理。编译器还执行语法和语义检查，并生成编译器错误和警告消息。

  汇编：在此步骤中，汇编器将汇编代码转换为可重定位目标代码。可重定位目标代码包含二进制表示形式的程序指令和数据，但还没有指定在内存中的位置。汇编器还会生成符号表，用于将程序中的符号（例如变量和函数名）映射到实际的内存地址。

  链接：在此步骤中，链接器将可重定位目标代码与其他必要的代码和库文件组合在一起，生成最终的可执行文件。链接器还解析符号引用，将程序中的符号引用与实际的内存地址相匹配。最终，可执行文件可以被操作系统加载到内存中，并执行其中的代码。

- 7）以下是一些常见的堆使用风险：
  内存泄漏：如果在堆上动态分配了一块内存，但是在程序执行过程中忘记了释放这块内存，就会导致内存泄漏，最终导致程序崩溃。

  指针悬挂：如果在堆上动态分配了一块内存，在指针指向这块内存之前，这块内存被释放了，那么指针就会悬挂，也会导致程序崩溃。

  缓冲区溢出：如果在堆上动态分配了一块内存，但是写入数据的时候超过了这块内存的边界，就会导致缓冲区溢出，也会导致程序崩溃。

  因此，当使用堆时，一定要注意内存管理，及时释放不再使用的内存，并且要确保操作堆的指针指向的内存是有效的。

- 8）MIPS 中的相对寻址（PC-relative addressing）模式：
  这种寻址模式是基于 PC 寄存器的当前地址计算出指令中的偏移量，从而得到操作数的地址。在相对寻址模式下，指令的操作数和指令本身在内存中的位置是有关系的。

  举个例子进行说明：假设我们有一条指令在地址 0x100 处，操作数偏移量为-4 个字节，那么操作数的地址就是 0x100-4=0xFC。在这种寻址模式下，偏移量通常是以 16 位的有符号数形式表示的。

  相对寻址模式的优点在于它可以使代码更加紧凑，因为它允许指令和操作数使用相对较小的偏移量进行寻址，而不需要使用全局地址。这样可以减少指令长度，从而提高代码的执行速度。缺点是在代码移动或修改时，相对地址可能会变得无效，从而导致代码出现问题。因此，在使用相对寻址模式时，需要格外小心以确保指令和操作数的相对位置不会受到意外的修改。

2. 算术运算

- 1）对于大量的加法操作，可以考虑使用并行加法器来提高效率。并行加法器可以同时进行多个加法运算，因此可以减少计算时间。常见的并行加法器有串行进位加法器和并行进位加法器两种。

  串行进位加法器是一种简单的并行加法器，其实现原理是将加法器分成多级，每级处理一个二进制位，从低位到高位依次进行加法计算。这种加法器的缺点是速度较慢，因为需要每次等待进位信号到达才能进行下一级的计算。

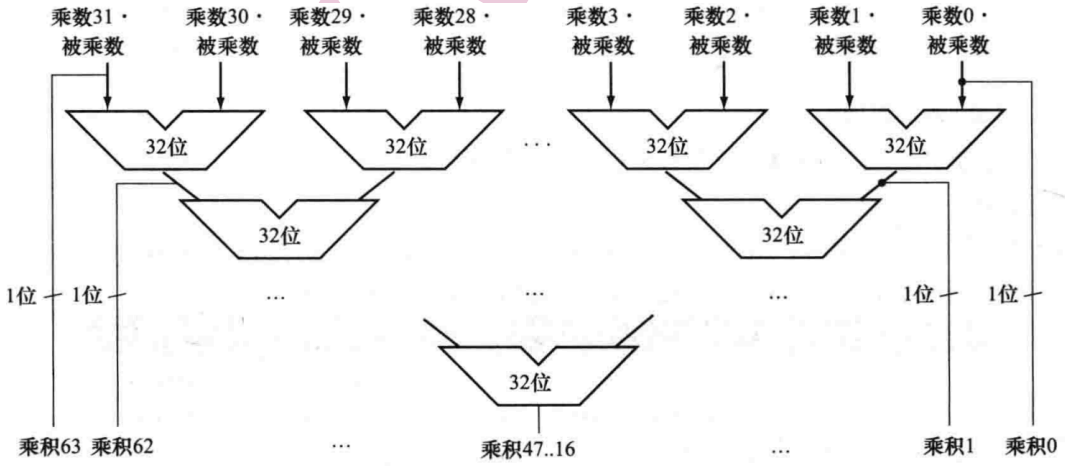
  并行进位加法器则可以更快地进行加法运算，因为它可以同时计算多个进位信号。常见的并行进位加法器有 Kogge-Stone 加法器和 Brent-Kung 加法器等。这些加法器的基本思想都是将加法器划分成多个部分，每个部分负责计算一个进位信号，然后将这些进位信号合并起来得到最终结果。

  使用并行加法器可以显著提高计算效率，但需要注意的是，这种方法通常需要较多的硬件资源和更复杂的电路设计。

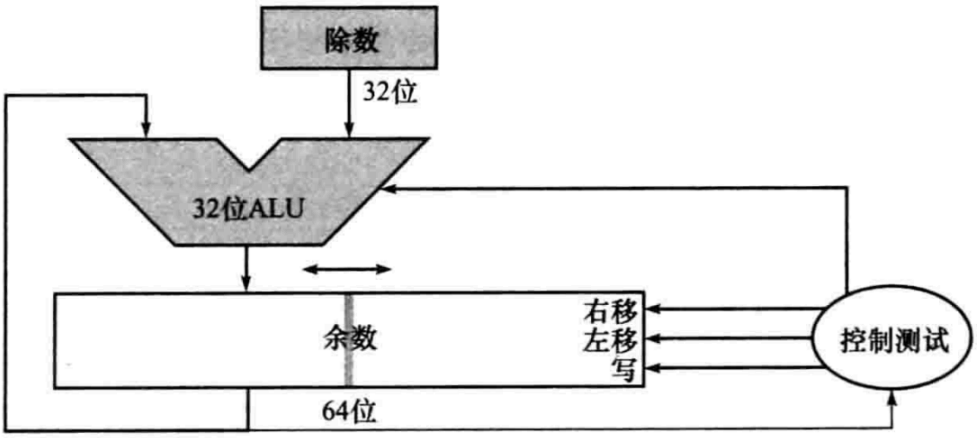
- 2）减法也可以通过加法来实现，他们有着相同的本质：减数在进行简单的取反之后再进行加法操作。

- 3）MIPS 采用两种类型的算术指令来解决这个问题：
  加法 (add) 立即数加法 (add 习和减法 (sub), 这三条指令在溢出时产生异常。
  无符号加法 (addu) 、立即数无符号加法 (add 江）和无符号减法 (subu), 这三条指令在发生溢出时不会产生异常。
  因为 C 语言忽略溢出，所以 MIPS C 编译器总是采用无符号的算术指令 addu addiu subu, 而不必考虑变量的类型但是 MIPS Fortran 编译器会根据操作数的类型来选择相应的算术指令。

- 4）下面是更快的乘法器的硬件结构：



  快速的乘法运算主要的思想是为乘数的位提供一个 32 位的加法器：一个用来输入被乘数和乘数位相与的结果，另一个是上个加法器的输出。然后将 32 个加法器组织成个并行树。
  下面是更快的除法器的硬件结构：



加速是通过将源操作数和商移位与减法同时进行注意到寄存器和加法器有未用的部分，可以通过将加法器和寄存器的位长减半来改进硬件结构如上图所示为改进后的硬件结构。

- 5）
  步骤 1: 不像加法，我们只是简单地将源操作数的指数相加来作为积的指数，现在我们处理带有偏阶的指数并要确定获得相同的结果，当将带偏阶的数相加时，为了得到正确的带偏阶的和，我们需要将一个偏阶从和中减去。
  步骤 2: 下面计算有效数的乘法。
  步骤 3: 这个积是未规格化的，所以我们需要规格化它。
  步骤 4: 因为之前我们假设有效数只有固定位宽（不包括符号），所以我们必须对结果进行舍入。
  步骤 5: 积的符号取决于原始源操作数的符号，当它们相同时，符号为正；否则，符号为负。

- 6）子字并行是一种数据并行的技术，它是指对一个字（通常是 32 位或 64 位）中的不同部分进行并行处理。例如，一个 32 位字可以被划分为 4 个 8 位的子字，这四个子字可以同时被处理。这种并行处理方式可以提高计算效率，尤其在大量数据的情况下，可以大幅度提高计算速度。
  针对上面的 while 循环，是不能 SIMD 并行化的，原因是：在这个 while 循环中，循环体内的操作不是独立的，后面的迭代依赖于前面的迭代。因此，这个 while 循环无法直接使用 SIMD 并行化。