**深 圳 大 学 实 验 报 告**

**课 程 名 称： 计算机系统(3)**

**实验项目名称： MIPS64乘法器模拟实验**

**学 院： 数学科学学院**

**专 业： 信息与计算科学（数学与计算机实验班）**

**指 导 教 师： 罗秋明**

**报告人： 王曦 学号： 2021192010 班级： 数计**

**实 验 时 间： 2023年10月07日**

**实验报告提交时间： 2023年10月07日**

**教务处制**

**一、 实验目标：**

实际运用WinMIPS64进行试验，以期更了解WinMIPS64的操作；

更加深入地了解MIPS程序的语法；

深入地了解在计算机中乘法的实现以及加法与乘法之间的关系。

**二、实验内容**

按照下面的实验步骤及说明，完成相关操作记录实验过程的截图：

首先，我们使用加法操作设计一个不检测溢出的乘法操作；完成后，我们对此进行优化，以期获得一个可以对溢出进行检测的乘法操作。（100分）

**三、实验环境**

硬件：桌面PC

软件：Windows，WinMIPS64仿真器

**四、****实验步骤及说明**

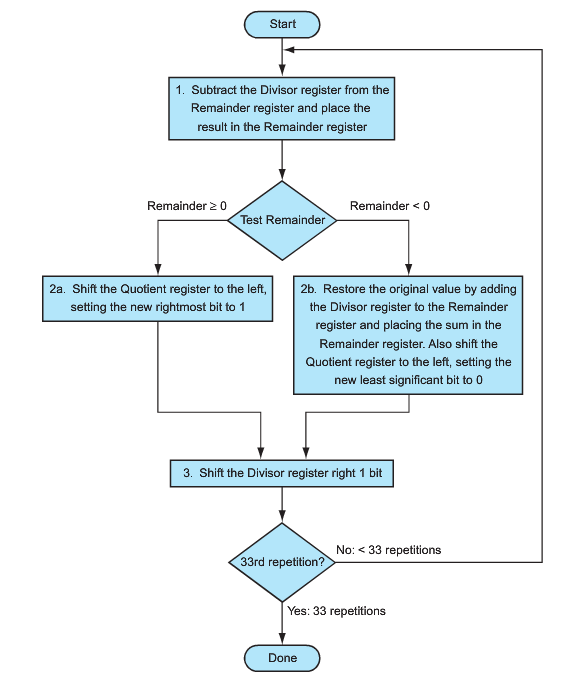
本次试验分为两个部分：第一部分、用加法器设计一个不考虑溢出的乘法器；第二部分、用加法器设计一个考虑溢出的乘法器（编程熟练的同学，也可以用除法器、浮点加法器等替代）。

1、忽略溢出的乘法器

首先，我们得了解乘法器如何由加法器设计得到，此处，我们以32位乘法为例。

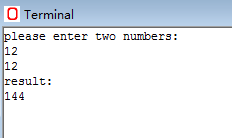
总共分为4步：

1. 测试乘数最低位是否为1，是则给乘积加上被乘数，将结果写入乘积寄存器；
2. 被乘数寄存器左移1位；
3. 乘数寄存器右移一位；
4. 判断是否循环了32次，如果是，则结束，否则返回步骤1。

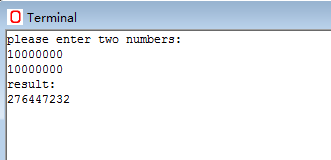


运行显示运行结果的例子如下，由于我们这里展示的是忽略了溢出的乘法，所以结果有两种：1、小于32位；2、大于32位。

第一种情况截图：



第二种情况截图：



根据上面的程序代码和截图，我们可以很清楚的看出，当结果小于32位时，结果正常；当结果大于32位时，结果只截取了低32位的结果，而高32位的结果直接忽略掉了。

**1.1 设置全局变量**

设置控制I/O的变量 CONTROL 和 DATA ，用于存放两个乘数的变量 NUM1 和 NUM2 ，栈大小 STACKSIZE 和栈顶指针 STACKPOINTER ，输出提示的字符串。

|  |
| --- |
| .data  CONTROL: .word 0x10000  DATA: .word 0x10008  NUM1: .word 0 # 乘数1  NUM2: .word 0 # 乘数2  STACK\_SIZE: .space 20 # 栈大小  STACK\_POINTER: .word 0 # 栈指针  INPUT\_STRING: .asciiz "please enter two numbers:\n"  RESULT\_STRING: .asciiz "result:\n"  WARNING\_STRING: .asciiz "warning: result overflow\n" |

**1.2 I/O函数**

**1.2.1 输出字符串的函数**

|  |
| --- |
| # writeString(a0, a1, a2)  # a0: 要输出的 string 的地址  # a1: DATA  # a2: CONTROL  writeString:  daddi $sp, $sp, -4 # 分配 1 个 int 的栈帧  sw $ra, ($sp) # M[sp] = ra , 保存返回地址    sw $a0, ($a1) # M[DATA] = 要输出的 string 的地址  daddi $t0, $zero, 4 # t0 = 4  sw $t0, ($a2) # M[CONTROL] = 4    lw $ra, ($sp) # ra = M[sp] , 恢复返回地址  daddi $sp, $sp, 4 # 回收栈帧  jr $ra # return |

**1.2.2 输入整数的函数**

|  |
| --- |
| # readInt(a0, a1, a2)  # a0: 要输入的 int 的地址  # a1: DATA  # a2: CONTROL  readInt:  daddi $sp, $sp, -4 # 分配 1 个 int 的栈帧  sw $ra, ($sp) # M[sp] = ra , 保存返回地址    # 输入一个 int 到 M[DATA]  daddi $t0, $zero, 8 # t0 = 8  sw $t0, ($a2) # M[CONTROL] = 8    # 将读入的 int 存到 M[a0]  lw $t1, ($a1) # t1 = M[DATA]  sw $t1, ($a0) # M[a0] = DATA    lw $ra, ($sp) # ra = M[sp] , 恢复返回地址  daddi $sp, $sp, 4 # 回收栈帧  jr $ra # return |

**1.2.3 输出整数的函数**

|  |
| --- |
| # writeInt(a0, a1, a2)  # a0: 要输出的 int 的地址  # a1: DATA  # a2: CONTROL  writeInt:  daddi $sp, $sp, -4 # 分配 1 个 int 的栈帧  sw $ra, ($sp) # M[sp] = ra , 保存返回地址    sw $a0, ($a1) # M[DATA] = 要输出的 int 的地址  daddi $t0, $zero, 2 # t0 = 2  sw $t0, ($a2) # M[CONTROL] = 2    lw $ra, ($sp) # ra = M[sp] , 恢复返回地址  daddi $sp, $sp, 4 # 回收栈帧  jr $ra # return |

**1.3 C/C++代码**

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  int main() {  cout << "please enter two numbers:" << '\n';    int num1, num2; cin >> num1 >> num2;  int ans = 0;  for (int i = 32; i; num1 >>= 1, num2 <<= 1, i--)  if (num1 & 1) ans += num2;    cout << "result:" << '\n';  cout << ans << '\n';  return 0;  } |

**1.4 主函数**

**1.4.1 初始化**

|  |
| --- |
| .text  main:  daddi $sp, $zero, STACK\_SIZE # 栈空间不超过 5 个 int    # 准备参数 DATA , CONTROL  lw $a1, DATA($zero) # a1 = M[0x10008]  lw $a2, CONTROL($zero) # a2 = M[0x10000] |

**1.4.2 输出提示字符串**

|  |
| --- |
| # 输出 INPUT\_STRING  daddi $a0, $zero, INPUT\_STRING # a0 = &INPUT\_STRING  jal writeString # call writeString() |

**1.4.3 输入两个乘数**

|  |
| --- |
| # 输入 NUM1, NUM2  daddi $a0, $zero, NUM1 # 读入 NUM1  jal readInt # call readInt()  daddi $a0, $zero, NUM2 # 读入 NUM2  jal readInt # call readInt() |

**1.4.4 求积**

|  |
| --- |
| # 求积  # t0: i  # t1: NUM1  # t2: NUM2  # t3: NUM1 的当前位  # t4: ans  daddi $t0, $zero, 32 # i = 32  lw $t1, NUM1($zero) # t1 = M[NUM1]  lw $t2, NUM2($zero) # t2 = M[NUM2]  loop:  beq $t0, $zero, endLoop # if (i == 0) goto endLoop;  andi $t3, $t1, 1 # t3 = NUM1 的当前位  beq $t3, $zero, notAdd # if (t3 == 0) goto notAdd;  dadd $t4, $t4, $t2 # ans += t2  notAdd:  dsrl $t1, $t1, 1 # NUM1 >>= 1  dsll $t2, $t2, 1 # NUM2 <<= 1  daddi $t0, $t0, -1 # i--  j loop |

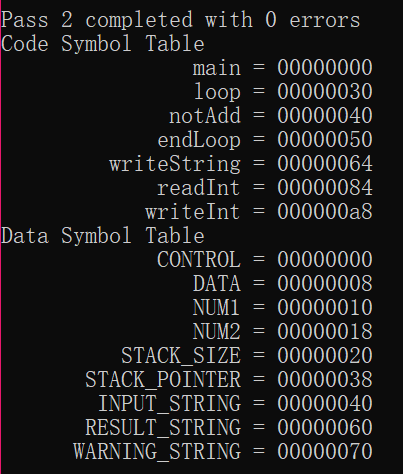
**1.4.5 输出答案（无溢出检测）**

|  |
| --- |
| endLoop:  # 输出 RESULT\_STRING  daddi $a0, $zero, RESULT\_STRING # a0 = &RESULT\_STRING  jal writeString # call writeString()    # 输出 ans  daddi $a0, $t4, 0 # a0 = ans  jal writeInt # call writeInt()    halt |

**1.5 运行结果**

**1.5.1 检查语法**

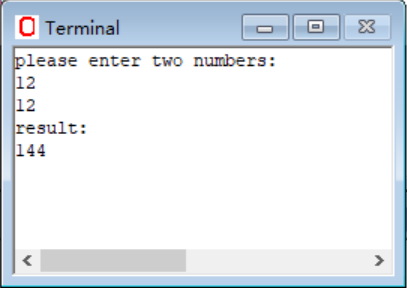
用 asm.exe 检查该程序的语法，发现无错误。



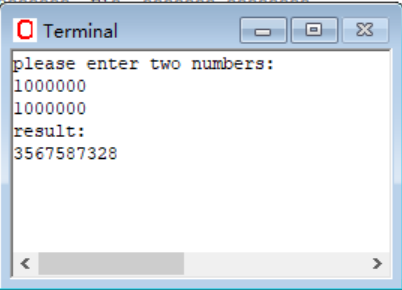
**1.5.2 运行结果**

将该程序载入 WinMIPS64.exe 并运行。

不发生溢出的情况如下图所示。



发生溢出的情况如下图所示。



2、溢出提示的乘法器

上述的程序，用加法实现了32位乘法，但是，其中，对溢出情况没有进行考虑是其中的弊端。这里，我们来完善上述的乘法器，使得该乘法器会在结果溢出时候提示。

其实，这个小优化是十分简单的，只需要对64位的寄存器中的高32位进行检测即可。当高32位为0时，说明结果没有溢出，否则，结果溢出。

**2.1 C/C++代码**

下面的代码用 long long 类型的变量表示一个 64 位的整数寄存器，并按上述规则检测溢出。

|  |
| --- |
| #include <bits/stdc++.h>  using namespace std;  int main() {  cout << "please enter two numbers:" << '\n';    int num1, num2; cin >> num1 >> num2;  long long ans = 0;  for (int i = 32; i; num1 >>= 1, num2 <<= 1, i--)  if (num1 & 1) ans += num2;    cout << "result:" << '\n';  cout << ans << '\n';  ans >>= 32;  if (ans) cout << "warning: result overflow" << '\n';  return 0;  } |

**2.2 溢出检测**

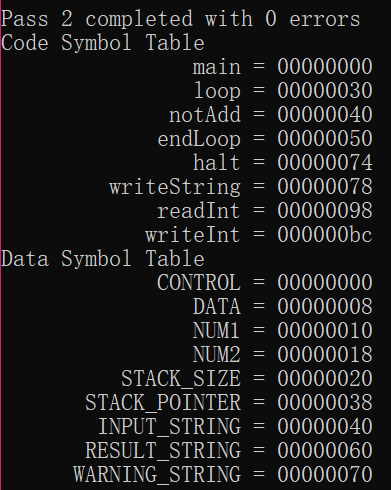
在输出结果和 halt 间加入溢出检测。

|  |
| --- |
| endLoop:  # 输出 RESULT\_STRING  daddi $a0, $zero, RESULT\_STRING # a0 = &RESULT\_STRING  jal writeString # call writeString()    # 输出 ans  daddi $a0, $t4, 0 # a0 = ans  jal writeInt # call writeInt()    # 溢出检测  dsrl $t4, $t4, 16 # ans >>= 16  dsrl $t4, $t4, 16 # ans >>= 16  beq $t4, $zero, halt # if (ans == 0) goto halt;    # 输出 WARNING\_STRING  daddi $a0, $zero, WARNING\_STRING # a0 = &WARNING\_STRING  jal writeString # call writeString()  halt:  halt |

**2.3 运行结果**

**2.3.1 检查语法**

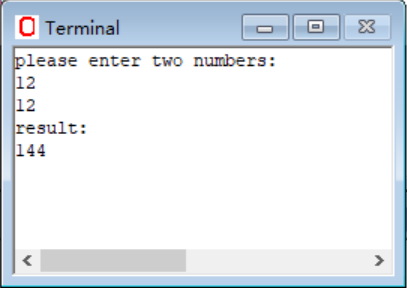
用 asm.exe 检查该程序的语法，发现无错误。



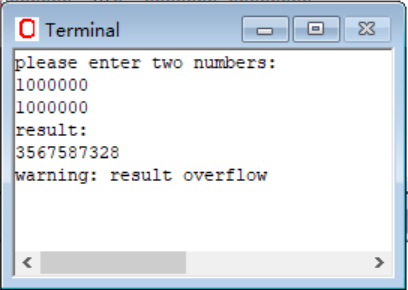
**2.3.2 运行结果**

将该程序载入 WinMIPS64.exe 并运行。

不发生溢出的情况如下图所示。



发生溢出的情况如下图所示。



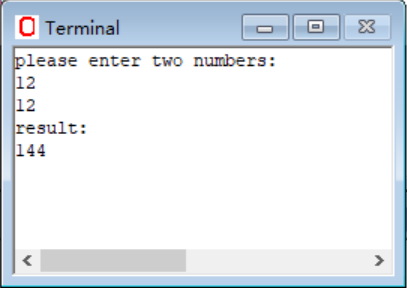
4 结束语

本实验介绍了通过加法器来设计乘法器的原理，并且在编写该实验程序的时候，我们更加了解了：1、计算机乘法器工作原理的内容；2、进一步熟练MIPS的编程方法；3、WinMIPS64的使用方法。当然，如果想要更加深入的学习，我们也可以课外继续编写对除法的模拟。Perf软件的使用让学生初步熟悉性能测评的主要工具。

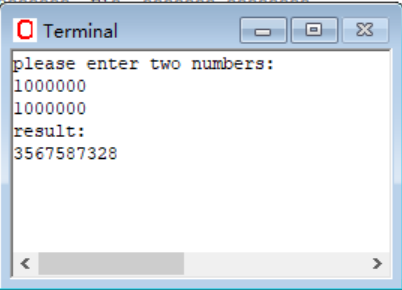
**五、实验结果**

**1. 无溢出检测的程序**

不发生溢出的情况如下图所示。

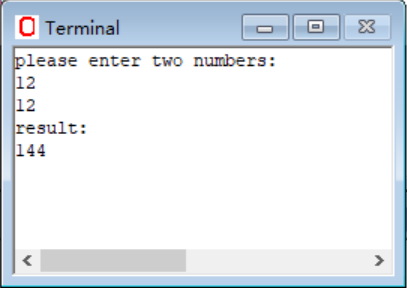


发生溢出的情况如下图所示。

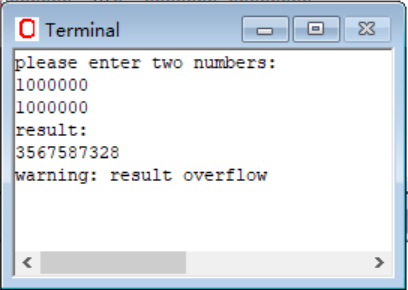


**2. 有溢出检测的程序**

不发生溢出的情况如下图所示。



发生溢出的情况如下图所示。



**五、实验总结与体会**

本次MIPS64乘法器模拟实验旨在通过实践运用WinMIPS64，加深对MIPS64程序语法和计算机中乘法实现方式的理解。实验内容主要包括设计不检测溢出的乘法操作和优化乘法操作以检测溢出。

首先，我们实现了一个简单的乘法操作，使用加法指令来模拟乘法运算，不考虑溢出情况。通过编写MIPS64汇编程序，我们成功实现了乘法操作，并将结果存储在指定的寄存器中。这一步骤使我们更熟悉MIPS64指令的使用和寄存器操作。

接着，我们优化了乘法操作，加入了对溢出情况的检测和处理。通过引入条件分支指令，我们能够在乘法运算后判断结果是否溢出，并根据情况采取相应措施。这个优化让我们更深入地理解了乘法运算可能遇到的问题以及如何通过程序逻辑解决这些问题。

通过这次实验，我们深入学习了MIPS64汇编语言中乘法运算的实现原理，同时也对计算机中乘法与加法之间的关系有了更加清晰的认识。同时，实践运用WinMIPS64使我们更加熟悉这一工具，为今后的计算机体系结构和程序设计提供了基础。在整个实验过程中，我们加深了对计算机原理的理解，并学到了如何将这些理论知识应用到实际问题中，这对于我们未来的学术研究和职业发展都具有重要意义。

|  |
| --- |
| **指导教师批阅意见：**  **成绩评定：**  指导教师签字：  年 月 日 |
| 备注： |