内存操作

简介

在高级语言中，我们可以使用多维数组对内存进行多维操作，但实际上，一般这些多维数组在内存中也只是按照一维的形式连续存储的。比如二维数组 arr[2][2]，他在内存中会占用 4 个单位的连续空间，分别保存 arr[0][0]、arr[0][1]、arr[1][0]、arr[1][1]。

作为低级语言，汇编语言对内存只能进行一维操作。而为了实现多维操作，我们就需要使用一些技巧了，下面会进行举例分析。

样例分析

比如我们要将 0~255 依次赋给一个 16*16 的矩阵，填充方式如下图 4*4 矩阵的例子。

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

使用高级语言的话只需要两个 for 循环嵌套，如下：

```
int value = 0;
int arr[16][16];
for (int i = 0; i < 16; i++) {
    for (int j = 0; j < 16; j++) {
        arr[i][j] = value++;
    }
}
```

使用汇编语言的话，就需要分为三步：

- 初始化数据

```
.data
    data: .word 0 : 256       # storage for 16x16 matrix of words
```

```
.text
    li $t0, 16          # $t0 = number of rows
    li $t1, 16          # $t1 = number of columns
    move $s0, $zero     # $s0 = row counter
    move $s1, $zero     # $s1 = column counter
    move $t2, $zero     # $t2 = the value to be stored
```

`$t0` 和 `$t1` 是总的行列数，`$s0` 和 `$s1` 是当前赋值的行列数，`$t2` 就是要赋的值。

- 为一行矩阵赋值

```
#________ tag1 ________
loop:
    mult $s0, $t1       # $s2 = row * #cols (two-instruction sequence)
    mflo $s2            # move multiply result from lo register to $s2
    add $s2, $s2, $s1   # $s2 += column counter
    sll $s2, $s2, 2     # $s2 *= 4 (shift left 2 bits) for byte offset
    sw $t2, data($s2)   # store the value in matrix element
#---------------------

#________ tag2 ________
    addi $t2, $t2, 1    # increment value to be stored
#---------------------

# Loop control: If we increment past last column, reset colume counter and increment
row counter
#               If we increment past last row,we're finished.

#________ tag3 ________
    addi $s1, $s1, 1    # increment coumn counter
#---------------------

    bne $s1, $t1, loop # not at end of row so loop back
```

tag1 代码段为当前矩阵元素对应的内存赋值，tag2 和 tag3 代码段更新数据，最后一行判断这一行矩阵是否已完全赋值。

- 准备为下一行矩阵赋值

```
    move $s1, $zero     # reset column counter
    addi $s0, $s0, 1    # increment row counter
    bne $s0, $t0, loop # not at end of matrix so loop back
# We're finished traversing the matrix
    li $v0, 10          # system service 10 is exit
    syscall             # we are outta here
```

当一行矩阵已赋完值后，更新数据，然后重新跳回到 loop 处为下一行矩阵赋值，直到整个矩阵都赋完值。

补充：利用Macros简化二维数组的计算

在 MIPS 教程部分，我们学习了如何使用 macro 宏定义一些操作来简化我们的 MIPS 程序，对于二维数组的相关计算，我们可以发现，计算数组中元素的地址是一个重复且固定的算法，因此，我们计划创建一个 macro 来代替这些语句，从而减少代码行数，增加可读性，以便出错后更好的调试代码。

```
.macro calc_addr(%dst, %row, %column, %rank)
    # dts: the register to save the calculated address
    # row: the row that element is in
    # column: the column that element is in
    # rank: the number of columns in the matrix
    multu %row, %rank
    mflo %dst
    addu %dst, %dst, %column
    sll %dst, %dst, 2
.end_macro
```

上图就是我们关于计算地址的 macro 了，我们精心的设计保证了运算中只改变 %dst 所代表的寄存器，避免了 macro 内部不可见的修改其他寄存器的情况的发生。这种复用代码的思想类似于函数，但是一定注意 macro 的运用中易出现的如上所述的问题，希望同学们能够熟练运用，写出易懂的汇编代码来。

思考题

结合一般程序的执行流，解释为什么 `MIPS` 指令中 link 类指令（如 `jal` 指令等），在不考虑延迟槽的情况下，将**当前指令地址 +4** 的值写入一个寄存器而不是写入**当前指令地址**或**当前指令地址 -4** 的值。

根据你编写过的汇编程序，总结程序中常见的**条件分支、循环与循环控制、函数调用、多维变量访问、格式化输入输出**等逻辑在汇编语言下的通用表达方式。

我们现在正在使用的计算机设备几乎全部都基于 `x86-64` 或 `ARM` 指令集，MARS 也运行在我们的电脑上，为什么其可以运行 `MIPS` 指令？结合相关资料解释原因。