Challenge！哈密顿回路

> 本题已给出C语言实现代码，我们要做的就是将其翻译为汇编语言，而难点在于如何将C语言中的递归算法转换为汇编语言中的循环算法。
> 下面我们将一步步地分析C语言代码，并给出相应的汇编代码。

定义数据段

```
matrix : .space 300      #邻接矩阵
book: .space 40          #标记数组
```

定义宏

在进入程序段之前，我们可以先利用宏对部分功能代码进行封装，便于实现代码复用，提高可读性。
在上一节 “函数调用 递归函数调用” 中，我们已经给出了几个常用的宏，可直接拷贝过来使用。

```
#程序结束
.macro end
    li      $v0, 10
    syscall
.end_macro
```

```
#读入整数
.macro get_int(%des)
    li      $v0, 5
    syscall
    move    %des, $v0
.end_macro
```

```
#打印整数
.macro print_int(%src)
    move    $a0, %src
    li      $v0, 1
    syscall
.end_macro
```

```
#入栈
.macro push(%src)
 addi    $sp, $sp, -4
```

```
 sw      %src, 0($sp)
.end_macro
```

```
#出栈
.macro pop(%des)
    lw      %des, 0($sp)
    addi    $sp, $sp, 4
.end_macro
```

```
#降维，并转化为相对于matrix的偏移量
.macro  getindex(%ans, %i, %j)
    sll %ans, %i, 3             # %ans = %i * 8
    add %ans, %ans, %j          # %ans = %ans + %j
    sll %ans, %ans, 2           # %ans = %ans * 4
.end_macro
```

观察整份C语言代码，注意到我们需要多次对两个数组中的元素进行赋值或判断，因此我们还可再定义两个宏，分别用于获取邻接矩阵和标记数组中元素的地址。

```
#获取matrix[x][y]的地址
.macro getmataddr(%des,%i, %j)
    getindex($t7, %i, %j)
    la %des,matrix
    add %des,%des,$t7
.end_macro
```

```
#获取book[x]的地址
.macro getbookaddr(%des,%src)
    sll $t7, %src, 2
    add %des, $s3, $t7
.end_macro
```

需要注意，使用宏时一定要谨慎。宏虽然方便，但使用不当会引发难以排查的错误。
尽管宏能像函数一样对代码起到封装复用的效果，但其本质是文本的替换。例如上述的两个宏，每次使用时都会改变`$t7`储存的数据，如果宏的上下文中都使用了`$t7`，那么可能产生错误。因此，在编写宏时，一定要确保宏的参数不会发生冲突，否则将会引发不可预知的问题。

程序段--main

```
.text
```

我们从main函数开始逐块翻译,首先我们将几个常用的值存入寄存器中，以便后续使用。同时我们要记住这几个已经使用的寄存器，避免重复使用。

```
main:
   get_int($s0)                        #$s0读入有向图G的结点数n
   get_int($s1)                        #$s1读入有向图G的边数m

   li $s2,1                            #$s2储存常量1
   li $s3,0                            #$s3储存结果，初始化为0
```

接下来循环读入邻接矩阵。对于只需要使用一次的变量，我们选择用寄存器$t0-$t7来存，它们储存临时变量，使用完后可以随时覆盖。（当然也可以使用$s0-$s7等其他寄存器，但我们为了便于区分哪些变量需要保留，哪些是一次性使用，我们就有了这个约定）

```
   li $t4,0                            #$t4为循环计数器，初始化为0
   loop1:
       beq $t4,$s1,  end_loop1

       get_int($t1)
       addi $t1,$t1,-1
       get_int($t2)
       addi $t2,$t2,-1

       getmataddr($t5,$t1,$t2)
       sw $s2, 0($t5)                  #将邻接矩阵matrix[x-1][y-1]置为1
       getmataddr($t5,$t2,$t1)
       sw $s2, 0($t5)                  #将邻接矩阵matrix[y-1][x-1]置为1

       addi $t4,$t4,1                  #循环计数器加1
       j loop1
   end_loop1:
```

我们先把main函数完成，之后再去写dfs函数的实现。
结点从0开始遍历，因此首先将0传给dfs函数。传参时，我们选择使用$a0-$a3寄存器，它们是函数调用的参数寄存器。

```
   li $t0,0                            #dfs从结点0开始

   move $a0,$t0                        #传参，$t0是实参，$a0是形参
   jal dfs                             #调用dfs函数

   print_int($s3)                      #输出结果
   end
```

接下来是本题的核心，dfs的递归调用

程序段--dfs

递归函数和普通函数的调用没有区别，最重要的都是的入栈和出栈，这是我们一定要深思熟虑的问题。
事实上，哪些数据的入栈和出栈，取决于我们希望函数调用结束前后，哪些数据需要被保留，哪些数据可以被覆盖。 我们只需要关注，**哪些寄存器中的变量，要在函数执行后，保持不变继续使用。**我们仅将这些变量入栈出栈即可（当然，也可以将函数内使用的所有寄存器全部入出栈，这样虽然麻烦，但是可以确保函数执行过程中不会对函数外造成影响）,同时ra是一定要入出栈的，不要忽视。
对于数据的出栈和入栈，我们可以在函数体写完后再去完成

```
    move $t0, $a0                   #将参数x存入临时寄存器t0中

    getbookaddr($t1,$t0)
    sw $s2, 0($t1)                  # 将book[x]置为1
```

```
#判断是否经过了所有的点

    li $t2, 0                       # $t2 = 0, 初始化循环变量
    li $t3,1                        # $t3 = 1, 作为标记flag
    loop2:
        beq $t2, $s0, end_loop      # 如果 $t2 == n, 则跳出循环
        getbookaddr($t4,$t2)
        lw $t4, 0($t4)

        and $t3,$t3,$t4             # $t3 &= book[$t2]
        addi $t2, $t2, 1            # $t2++
        j loop2                     # 跳转到循环开始
    end_loop:
```

下面的条件判断是两个用&&连接的语句，在C语言里表现为**短路效应**，即如果第一个条件不满足，则不会判断第二个条件，因此我们在使用汇编语言时，先判断第一个条件，如果第一个条件不满足，则直接跳转到else语句，否则再判断第二个条件。
注意到如果条件成立将会执行return，即函数结束，因此我们需要在函数结束前将寄存器出栈，否则将会导致函数无法正常返回。由于我们还未确定哪些寄存器入出栈，可以先在此标记，最后再来补全

```
#判断是否满足哈密顿回路条件

    beq $t3, $s2 ,if_1
    else_1:
        j end_if_else12
    if_1:
        getmataddr($t1,$t0,$zero)
        lw $t1,0($t1)
        beq $t1,$s2,if_2
    else_2:
        j end_if_else12
    if_2:
        li $s3 ,1
```

```
        ####
        ####return
        ####

        jr $ra
    end_if_else12:
```

```
#搜索与之相邻且未经过的边

    li $t1,0                        # $t1 = 0，初始化循环变量
    loop3:
        beq $t1, $s0, end_loop3     # 如果 $t1 == n，则跳出循环

        getbookaddr($t2,$t1)
        lw $t2, 0($t2)
        getmataddr($t3,$t0,$t1)
        lw $t3,0($t3)

        beq $t2,$zero,if_3
        else_3:
            j end_if_else34
        if_3:
            beq $t3,$s2,if_4
        else_4:
            j end_if_else34
        if_4:
            move $a0, $t1
            jal dfs
        end_if_else34:
        addi $t1, $t1, 1            # $t1++
        j loop3                     # 跳转到循环开始
    end_loop3:
```

```
# 将book[x]置为0

    getbookaddr($t1,$t0)
    sw $zero, 0($t1)
```

函数体已经完成，下面补全出栈入栈部分。
我们需要关注dfs函数执行完毕后还在使用的寄存器。
首先是main函数部分，执行完dfs后，只有储存最终结果ans的寄存器即$s3需要再使用，但它本身就是通过dfs来赋值的，所以无需入出栈

```
    // 从第0个点（编号为1）开始深搜
    dfs(0);
    printf("%d", ans);
    return 0;
}
```

然后是dfs函数对自己的递归调用,i和x要保证在dfs前后保持不变，对应的寄存器是$t1和$t0

```
    for (i = 0; i < n; i++) {
        if (!book[i] && G[x][i]) {
            dfs(i);
        }
    }
    book[x] = 0;
}
```