实验平台说明

特征	说明
操作系统版本和系统类型	Windows 10 64 bit
Quartus 版本	Quartus Prime 13.1.0
验证方式	Simnios 仿真

自查清单

部分	完成度
Part1	100%
Part2	100%
Part3	100%
Part4	100%
Part5	100%

第一二部分　Part 1part2/Lab3

一.题目分析

由于 part1 部分比较简单，前两个实验部分常涉及，所以这里跟 part2 做在一起。Part1 中使用 Quartus ii 建立一个的 nios ii 系统。使用 qsys 并包含以下部分，JTAG,RAM。然后自动分配器件中的基地址。生成系统后，在模块中例化生成的 nios ii 系统。查询到正点原子 FPGA 教程中有 nios ii 中生成系统的详细步骤，这里只需要按照既定步骤建立即可。

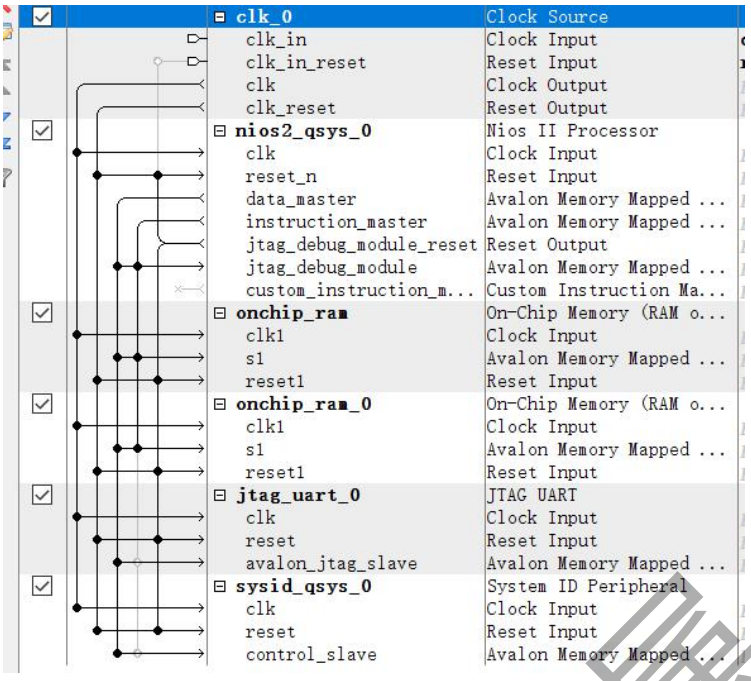
Part2 中，先把文件 file 加载到储存器中，然后对储存的数据进行排序即可。在本 part 使用冒泡排序法。

二.实验过程

1.类似于 lab1 中的 part1 部分。

clk_0	Clock Source			
clk_in	Clock Input	**clk**	*exported*	
clk_in_reset	Reset Input	**reset**		
clk	Clock Output	Double-click to	clk_0	
clk_reset	Reset Output	Double-click to		
nios2_qsys_0	Nios II Processor			
clk	Clock Input	Double-click to	**clk_0**	
reset_n	Reset Input	Double-click to	[clk]	
data_master	Avalon Memory Mapped ...	Double-click to	[clk]	IRQ 0
instruction_master	Avalon Memory Mapped ...	Double-click to	[clk]	
jtag_debug_module_reset	Reset Output	Double-click to	[clk]	
jtag_debug_module	Avalon Memory Mapped ...	Double-click to	[clk]	**0x0001_8800**
custom_instruction_m...	Custom Instruction Ma...	Double-click to		
onchip_ram	On-Chip Memory (RAM o...			
clk1	Clock Input	Double-click to	**clk_0**	
s1	Avalon Memory Mapped ...	Double-click to	[clk1]	**0x0000_8000**
reset1	Reset Input	Double-click to	[clk1]	
onchip_ram_0	On-Chip Memory (RAM o...			
clk1	Clock Input	Double-click to	**clk_0**	
s1	Avalon Memory Mapped ...	Double-click to	[clk1]	**0x0001_4000**
reset1	Reset Input	Double-click to	[clk1]	
jtag_uart_0	JTAG UART			
clk	Clock Input	Double-click to	**clk_0**	
reset	Reset Input	Double-click to	[clk]	
avalon_jtag_slave	Avalon Memory Mapped ...	Double-click to	[clk]	**0x0001_9008**
sysid_qsys_0	System ID Peripheral			
clk	Clock Input	Double-click to	**clk_0**	
reset	Reset Input	Double-click to	[clk]	
control_slave	Avalon Memory Mapped ...	Double-click to	[clk]	**0x0001_9000**

2.连接器件并将端口引出，这里参考了正点原子的方法，双击并起名。



3.自动分配地址。

4.编写.V 文件，例化系统。

5 编写代码

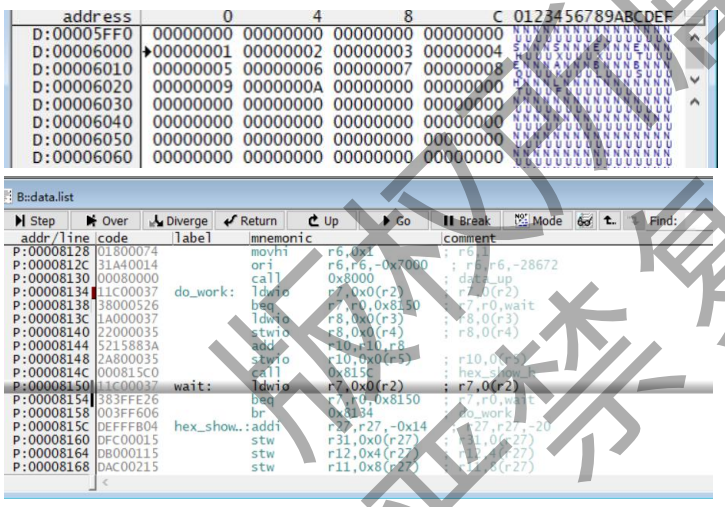
6.加载 file 文件，由于实验条件原因，这里只是参考了吴雨林同学的加载方法。将 HEX6 加载到起始地点 0x00006000

建立

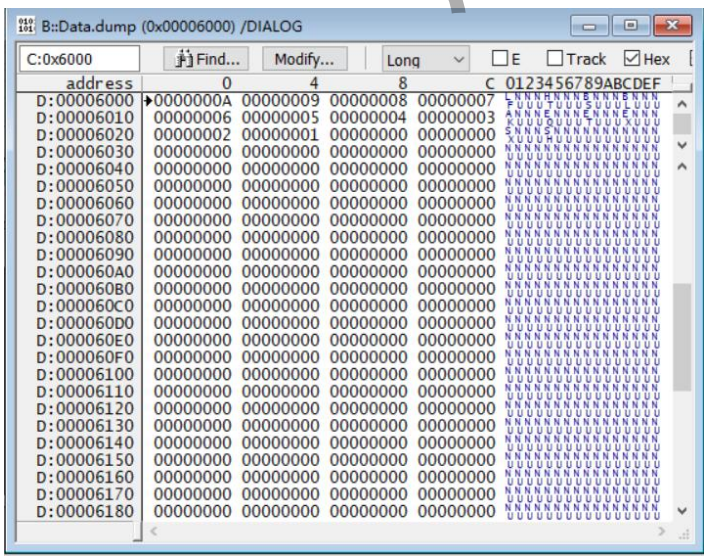
E:/altera/13.1/lab/qrs_lab3/qrs_lab3.v Hex6.he

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
0	0	0	0	10	0	0	0	1	.....
8	0	0	0	2	0	0	0	3	.....
16	0	0	0	4	0	0	0	5	.....
24	0	0	0	6	0	0	0	7	.....
32	0	0	0	8	0	0	0	9	.....
40	0	0	0	0	0	0	0	0	.....
48	0	0	0	0	0	0	0	0	.....
56	0	0	0	0	0	0	0	0	.....
64	0	0	0	0	0	0	0	0	.....
72	0	0	0	0	0	0	0	0	.....

7.仿真



正确排序



三.代码详解

```
.include "nios_macros.s"
.text
.global _start

.equ LIST_DATA, 0x00006000   //数据储存地址

_start:
    movia r2,LIST_DATA //将地址加载到 r2 中
    addi r3,r2,4       //将头数据地址加载到 r3 中
    ldw r4,(r2)        //将 r2 的内的值加载到 ldw 即总排序个数。
BUBBLE:
    movi r5,1          //r5 储存 1，作为 r6 和 r7 的参照
    mov r6,r4          //r6 为内循环中的循环次数
    movi r7,0          //r7 为内循环中的循环次数
INIT_LOOP:
LOOP:
    beq r6,r5,END_LOOP//相等则转移结束
    subi r6,r6,1       //r6 自减一
    br SWAP
END_LOOP:
    subi r4,r4,1
    mov r6,r4
    addi r3,r2,4
    beq r7,r5,INIT_LOOP   /*r7=1 则继续循环，因为这里从大到小排序，则最后一个
                          数不需排列，则不用进入 CHANGE，或者可以这样理解，最
                          后一次循环，没有需要交换的数，所以 r7=0*/
END:
    br END
SWAP:
    ldw r8,(r3)        //r8，r9 作为交换的缓冲寄存器
    ldw r9,4(r3)       //先行载入
    blt r8,r9,CHANGE   //若小于则转移。
END_SWAP:
    addi r3,r3,4
    br LOOP
CHANGE:
    stw r8,4(r3)       //交换载入。
    stw r9,(r3)
    movi r7,0x1        //使能 r7
```

```
    br END_SWAP        //结束
.end
```

四.组长点评

本实验难点在于理解冒泡排序法在汇编中的编程方法。这里可以将寄存器列在一个表格中，并把常用的地址也标注出。进行

第三部分　Part 3/Lab3

一.题目分析

Part3 要求我们使用子程序让子程序更加通用，即设定个数和初地址在某个寄存器，然后使用子程序完成各种移动，并将寄存器的值储存在栈中，以期现场免遭破坏。

二.实验过程

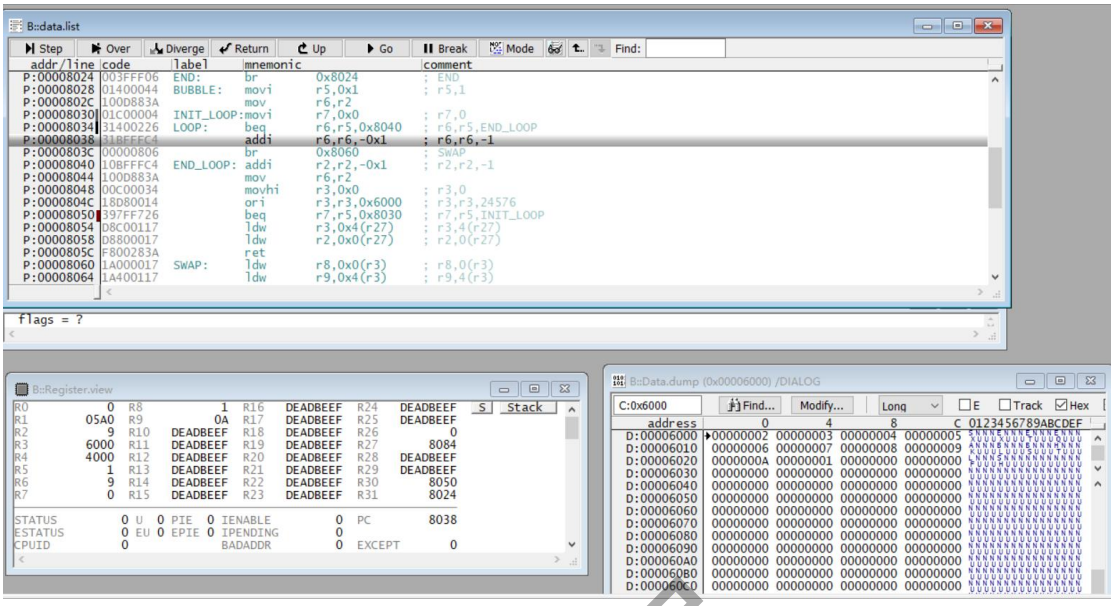
1.编写代码

2.建立汇编工程。

3.加载 file 文件，由于实验条件原因，这里只是参考了吴雨林同学的加载方法。将 HEX6 加载到起始地点 0x00006000

建立

4.仿真



正确排序

三.代码解释

```
.include "nios_macros.s"
.text
.global _start
                        //除现场保护外，其他部分与 part2 一样。
.equ LIST_DATA, 0x00006000 //数据储存地址

_start:
    movia sp,SP_INIT
    movia r3,LIST_DATA
    movia r2,10
    stw r2,0(sp)        //在此保护寄存器 r2 现场
    stw r3,4(sp)        //保护寄存器 r3 现场，在结束时恢复
    call BUBBLE
END:
    br END

BUBBLE:
    movi r5,1
    mov r6,r2
INIT_LOOP:
    movi r7,0
LOOP:
    beq r6,r5,END_LOOP
    subi r6,r6,1
```

```
    br SWAP
END_LOOP:
    subi r2,r2,1
    mov r6,r2
    movia r3,LIST_DATA
    beq r7,r5,INIT_LOOP
    ldw r3,4(sp)      //恢复 r3 现场
    ldw r2,0(sp)      //恢复 r2 现场
    ret
SWAP:
    ldw r8,(r3)
    ldw r9,4(r3)
    blt r8,r9,CHANGE
END_SWAP:
    addi r3,r3,4
    br LOOP
CHANGE:
    stw r8,4(r3)
    stw r9,(r3)
    movi r7,1
    br END_SWAP
SP_INIT:.skip 200
.end
```


四.组长点评

Part3 主要是考虑一个保护现场的问题，首先在 C 语言的语法里，保存一个数据，并对一个数据进行一个处理。该需要初始化并使用 sp 寄存器，这里不是特别难。

第四部分　Part 4/Lab3

一.题目分析

Part3 要求从主程序到子程序传递参数，这里我理解的是在转移 PC 地址时，对寄存器内的数据进行保护，由于 ram 储存不稳定（当然是相对于 rom 而言）。所以这里需要用 sp，并压入栈中。

二.实验过程

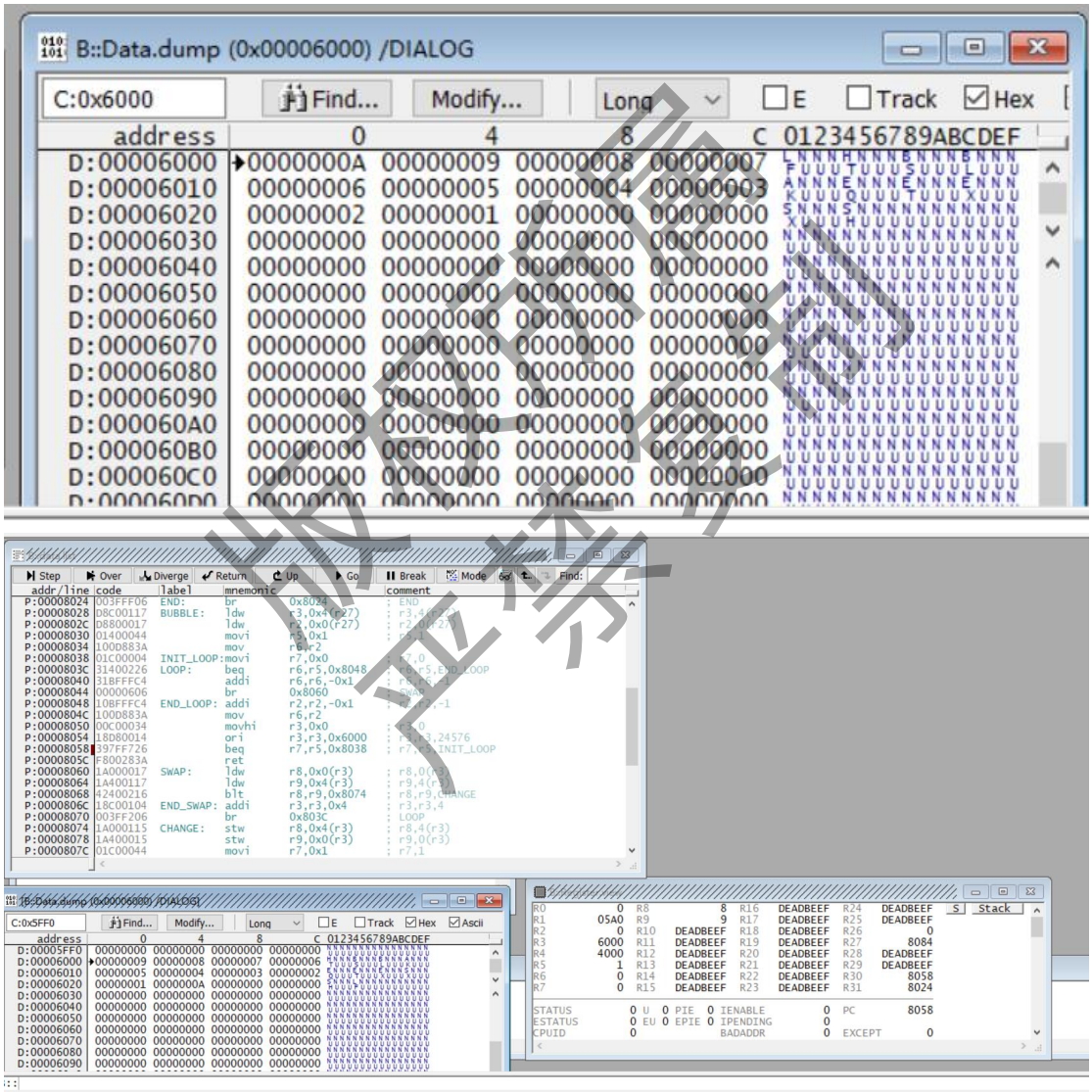
1.编写代码

2.建立汇编工程。

3.加载 file 文件，由于实验条件原因，这里只是参考了吴雨林同学的加载方法。将 HEX6 加载到起始地点 0x00006000

建立

4.仿真



正确排序

三.代码详解

```
.include "nios_macros.s"
.text
.global _start
                        //除现场保护外，其他部分与 part2 一样。

.equ LIST_DATA, 0x00006000

_start:
    movia sp,SP_INIT
    movia r3,LIST_DATA
    movia r2,10
    stw r2,0(sp)      //保护寄存器 r2 的值，压入栈中
    stw r3,4(sp)      //保护寄存器 r3 的值，压入栈中
    call BUBBLE
END:
    br END

BUBBLE:
    ldw r3,4(sp)      //恢复寄存器的值
    ldw r2,0(sp)      //恢复寄存器的值
    movi r5,1
    mov r6,r2
INIT_LOOP:
    movi r7,0
LOOP:
    beq r6,r5,END_LOOP
    subi r6,r6,1
    br SWAP
END_LOOP:
    subi r2,r2,1
    mov r6,r2
    movia r3,LIST_DATA
    beq r7,r5,INIT_LOOP
    ret
SWAP:
    ldw r8,(r3)
    ldw r9,4(r3)
    blt r8,r9,CHANGE
END_SWAP:
    addi r3,r3,4
    br LOOP
CHANGE:
```

```
    stw r8,4(r3)
    stw r9,(r3)
    movi r7,1
    br END_SWAP
SP_INIT:.skip 200
.end
```

四.组长点评

Part4 的内容时对主程序到子程序的参数传递，为了安全性，这里使用 sp 寄存器联通栈储存并转移。相对来说比较简单。

第五部分　Part 5/Lab3

一.题目分析

Part5 的实验内容为使用嵌套程序计算阶乘。这里写出 C 语言的程序。

```
long f(long n);
long f(long n){
    if(n ==1){判断是否等于 1
       return 1;是则返回
    }else{
        return f(n-1)*n; 否则储存地址及储存数据
   }
```

按照这个思路进行汇编代码的编写。参考吴雨林同学的设计，一步一步推演寄存器反应。

				sp	
r4	r5	ra	r8	0x0000	0x00001000
2	0	0x00001000	2	0x0004	2
1	0	0x00002000		0x0008	0x00002000
0	0	0x00003000		0x0012	1
0	1	0x00003000		0x0016	0x00003000
1	1	0x00002000		0x0020	0
2	2	0x00001000		0x0024	
		END			

基本思路就是在栈中储存数据及转移地址。

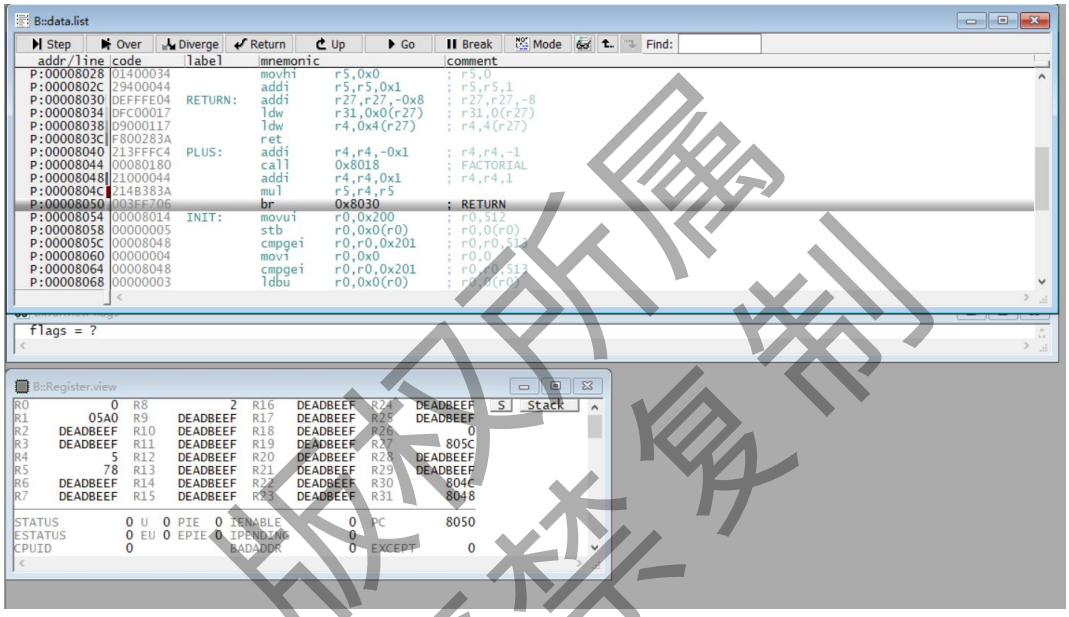
二.实验过程

1.编写代码

2.建立汇编工程。

3.加载 file 文件，由于实验条件原因，这里只是参考了吴雨林同学的加载方法。将 HEX6 加载到起始地点 0x00006000

4. 建立仿真



测试 5，结果 0x78，正确。

三.代码分析

代码参考了吴雨林同学的代码

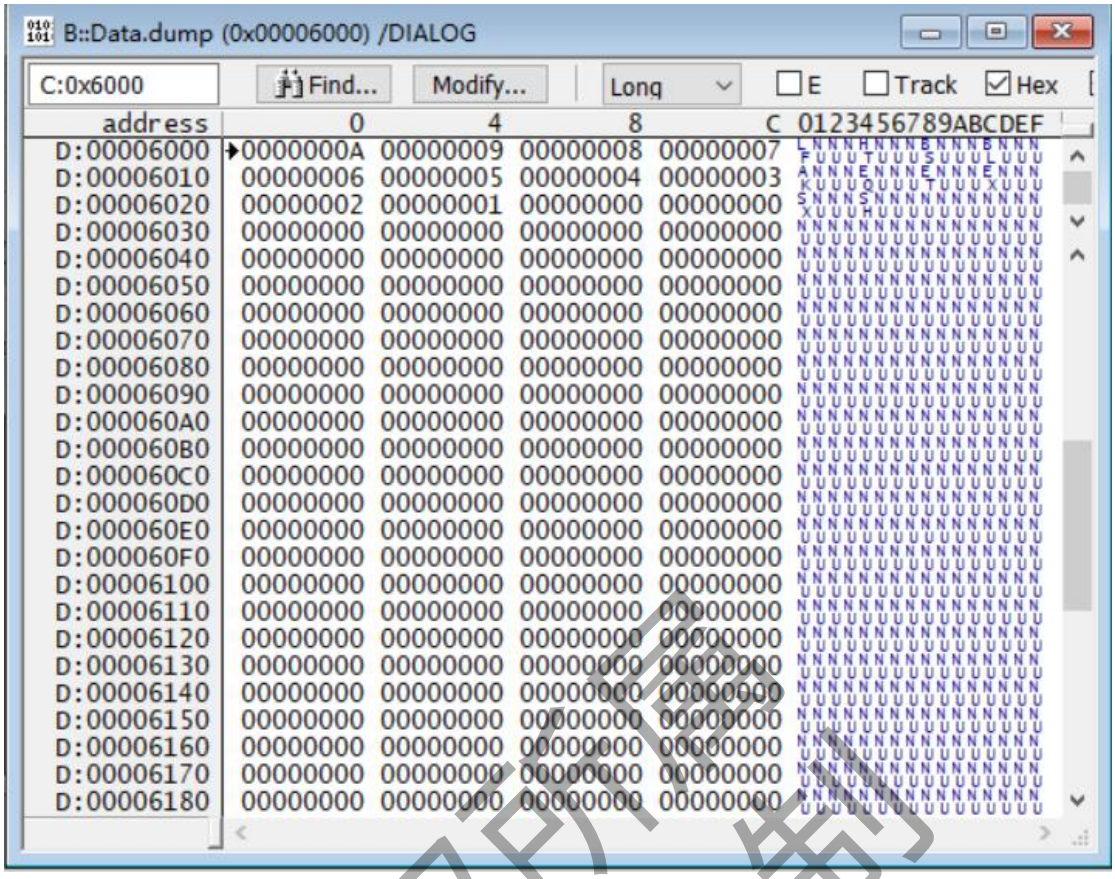
```
.text
.global _start

_start:
    movi r4,5
    movia sp,INIT      //sp 初始化
    call FACTORIAL     //调用子程序
    END: br END
FACTORIAL:
    stw ra,(sp)        //储存转移地址
```

```
    stw r4,4(sp)   //储存数据
    addi sp,sp,8   //栈地址+8，为下一次储存做准备
    bne r4,r0,PLUS     //如果不是 0，则转移到 PLUS，为嵌套准备。
    movia r5,1     //若为 0，当前嵌套结束
RETURN:
    subi sp,sp,8   //sp-8 以返回上次储存处
    ldw ra,(sp)    //载入上次嵌套地址
    ldw r4,4(sp)   //载入上次嵌套阶乘的预备数
    ret
PLUS:
    subi r4,r4,1   //嵌套次数或者称阶乘次数-1
    call FACTORIAL//开始嵌套。
    addi r4,r4,1   //嵌套次数+1
    mul r5,r4,r5   //r4，r5 相乘并储存
    br RETURN      //嵌套层返回
INIT:.skip 400
.end
```

四.组长点评

本实验难度在于返回 ra 的节点，这个转移的节点把握好就可以。其次就是嵌套次数计数，预备数的处理做好储存即可。程序逻辑性比较困难，我选择使用寄存器来进行人工推演。这样的话就比较简单明显易懂。


B::Data.dump (0x00006000) /DIALOG
C:0x6000
Find...
Modify...
Long
E
Track
Hex
address 0 4 8 C 0123456789ABCDEF
D:00006000 0000000A 00000009 00000008 00000007
D:00006010 00000006 00000005 00000004 00000003
D:00006020 00000002 00000001 00000000 00000000
D:00006030 00000000 00000000 00000000 00000000
D:00006040 00000000 00000000 00000000 00000000
D:00006050 00000000 00000000 00000000 00000000
D:00006060 00000000 00000000 00000000 00000000
D:00006070 00000000 00000000 00000000 00000000
D:00006080 00000000 00000000 00000000 00000000
D:00006090 00000000 00000000 00000000 00000000
D:000060A0 00000000 00000000 00000000 00000000
D:000060B0 00000000 00000000 00000000 00000000
D:000060C0 00000000 00000000 00000000 00000000
D:000060D0 00000000 00000000 00000000 00000000
D:000060E0 00000000 00000000 00000000 00000000
D:000060F0 00000000 00000000 00000000 00000000
D:00006100 00000000 00000000 00000000 00000000
D:00006110 00000000 00000000 00000000 00000000
D:00006120 00000000 00000000 00000000 00000000
D:00006130 00000000 00000000 00000000 00000000
D:00006140 00000000 00000000 00000000 00000000
D:00006150 00000000 00000000 00000000 00000000
D:00006160 00000000 00000000 00000000 00000000
D:00006170 00000000 00000000 00000000 00000000
D:00006180 00000000 00000000 00000000 00000000