

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%
Part6	100%

第一部分 Part 1/Lab1

一.题目分析

使用 Quartus ii 建立一个简单的 nios ii 系统。使用 qsys 并包含以下部分，JTAG,32KRAM。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。搜索到正点原子 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。

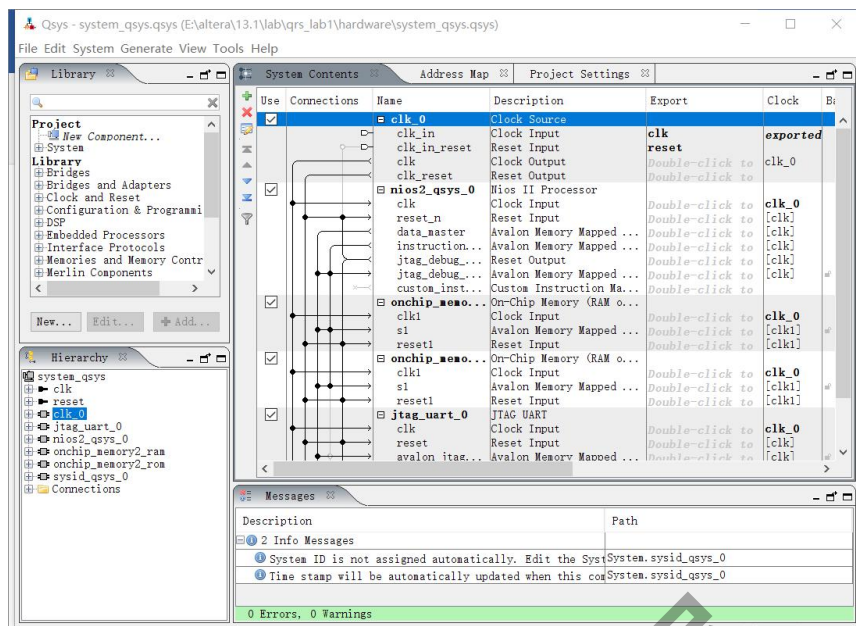
二 . 实验过程

1.首先建立一个 quatus ii 的工程。

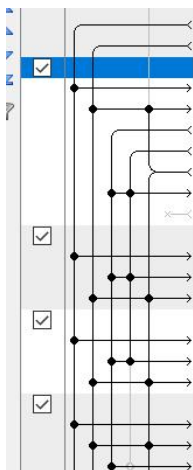


2.点击 Qsys，查看并修改 clk 的频率到 50MHZ。

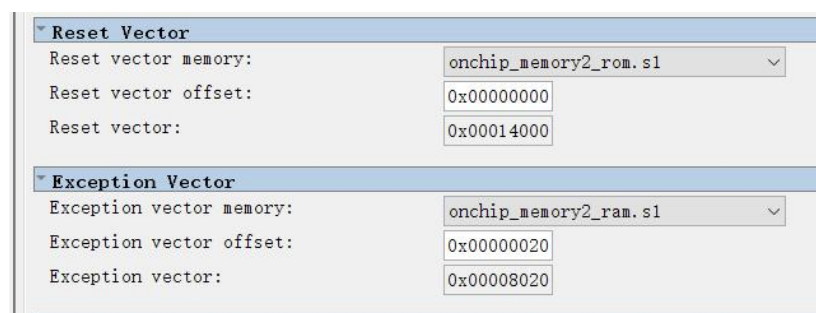
3.按照正点原子的步骤添加 nios ii, JTAG,RAM, System ID IP 等如下图。



4.连线，将每个模块的 clk，rst 等线连接。这里连线因为不熟悉，参考正点原子。

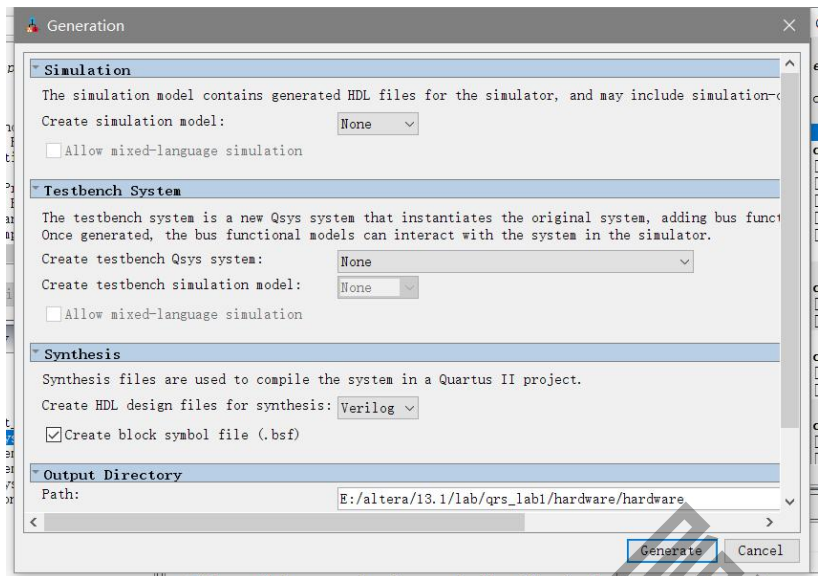


5.修改自动复位和异常复位如图所示，并自动分配基地址。他们代表了在 FPGA 中的位置。

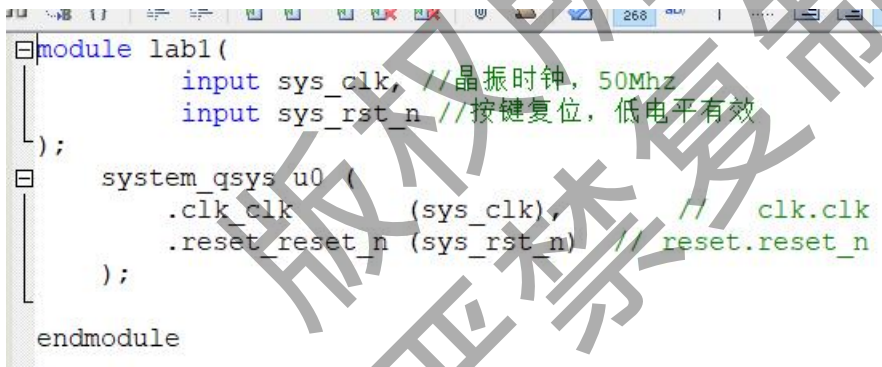


6. 点击菜单栏中的【Generate】→【Generate...】选项，选择语言，

安装位置等，生成系统。



7.在工程文件中建立一个.v 文件，并点击 Qsys 菜单栏的【Generate】→【HDL Example】，赋值初始化代码。在.V 文件中例化如图所示。



8.编译通过，由于手头没有板子，进行仿真可以不需分配引脚。LAB1 结束。

三 . 代码详解

该部分无代码。

四 . 组长点评

Part1 内容较少，比较简单，只要跟随正点原子开源例程即可完美完成。其中难点在于连线问题比较困难，这个是需要累积的部分，初期

只能简单模仿。

第二部分 Part 2/Lab1

一.题目分析

Part2 主要内容为使用提供的代码在实际的开发板上进行，受实验条件的限制，本次实验必须以仿真方式进行。提供的代码为汇编语言对 0x90abcdef 中连续出现的 1 的最大长度。具体的代码分析在三中。

二.实验过程

1. 使用 vs code 编辑 lab.s

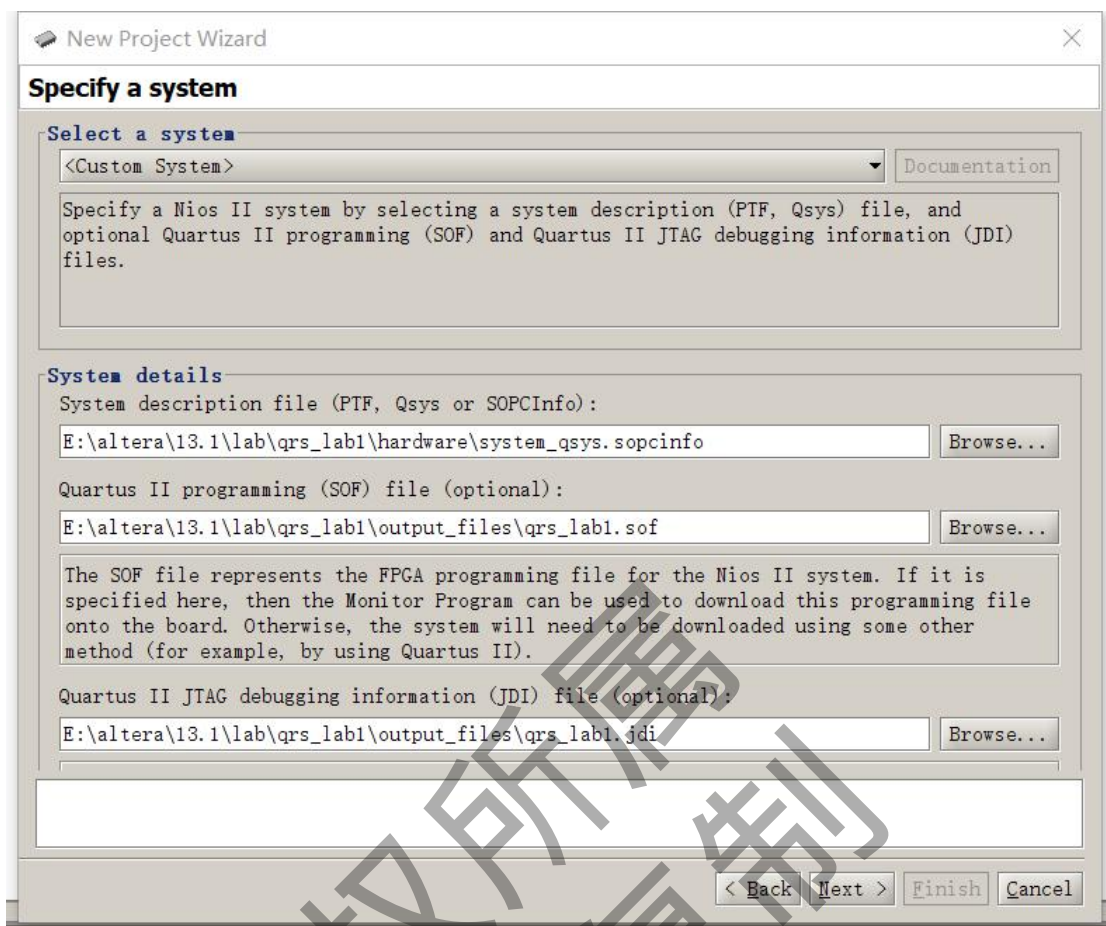
```
.include "nios_macros.s"
.text
.equ TEST_NUM,0x90abcdef

.global _start
_start:
    movia r7,TEST_NUM
    mov r4,r7
    STRING_COUNTER:
    mov r2,r0
    STRING_COUNTER_LOOP:
    beq r4,r0,END_STRING_COUNT
    srli r5,r4,0x01
    and r4,r4,r5
    addi r2,r2,0x0001
    br STRING_COUNTER_LOOP

    END_STRING_COUNTER:
    mov r16,r2

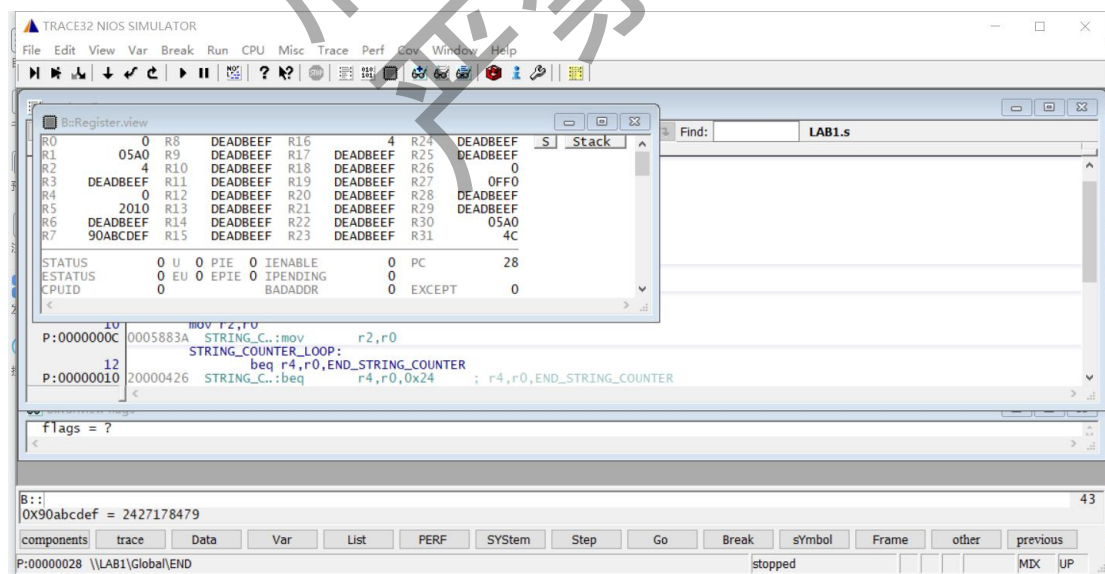
    END:
    br END
.end
```

2. 使用软件 altera monitor program.建立工程，系统可以选择 part1 生成，也可以选择使用自带的 de2 系统。



3. 将 lab.s 加入工程中，编译。

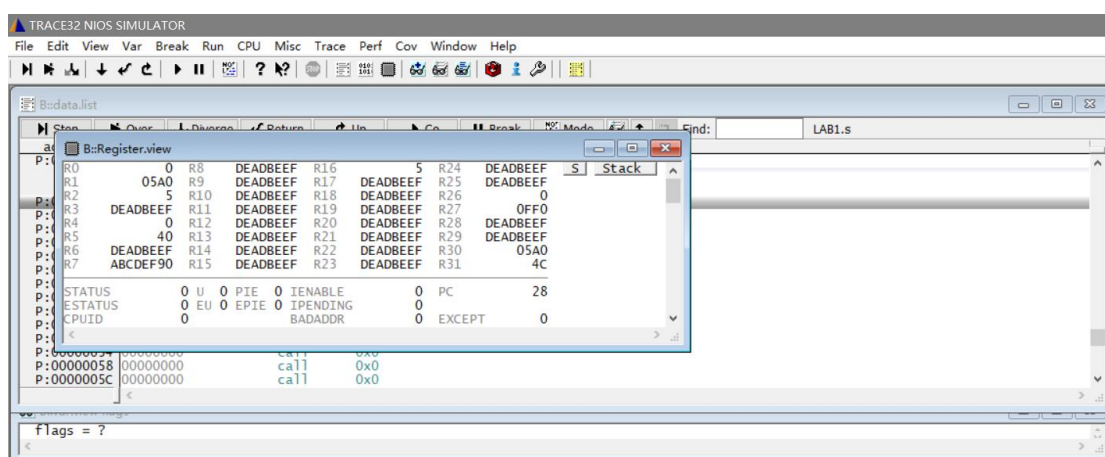
4. 打开 Simnios，加载 lab.elf 文件如图



5. 运行程序发现 R16 的值为 4。

6. TEXT 改为 0xabcdef90,然后重新重复上述过程。发现 R16 为 5,

则 part2 完成。



三.代码分析

该部分代码为汇编语言，据该部分的注释部分可以简要分析此代码。首先定义 `text_num` 为 `0x90abcdef`，程序主体从 `start` 开始，将定义的符号数赋给 `r7` 寄存器，再将 `r7` 的值赋给 `r4`，进入 `STRING_COUNTER`，将 `r0` 的值赋给 `r2`，这相当于将 `r2` 清零，`STRING_COUNTER1` 包含一个循环，循环的名字是 `STRING_COUNTER_LOOP`，`beq` 指令比较 `r4,r0` 寄存器中的内容，相同则跳转到 `END_STRING_COUNTER`，否则程序向下执行先判断再执行循环体。循环体内部，使用 `srlr` 将 `r4` 的值补 0 右移一位，结果存在 `r5` 中，使用 `and` 将 `r5` 和 `r4` 按位与，结果存在 `r4` 中，`addi` 使 `r2` 中的结果加 1，最后 `br` 直接跳转到 `STRING_COUNTER_LOOP` 判断条件。循环结束时跳转到 `END_STRING_COUNTER`，使用 `mov` 将 `r2` 的值赋给 `r16`。程序结束。

主要用到的最核心的东西是若一个数有连续的 1，则这个数与其本身的数往右移一位的数按位与之不为 0。

```

.include "nios_macros.s"
.text
.equ TEST_NUM,0x90abcdef

.global _start
_start:
    movia r7,TEST_NUM
    mov r4,r7
STRING_COUNTER:
    mov r2,r0
STRING_COUNTER_LOOP:
    beq r4,r0,END_STRING_COUNT
    srli r5,r4,0x01
    and r4,r4,r5
    addi r2,r2,0x0001
    br STRING_COUNTER_LOOP

    END_STRING_COUNTER:
    mov r16,r2

    END:
    br END
.end

```

四.组长点评

本个实验主要是对于 niosii 开发仿真流程操作进行测试，在有部分同学开源的情况下，并不是很复杂。主要难点在于对程序的理解，需要自己手动演算，才能得出结论。实际上操作并不是很麻烦。

一． 题目分析

Part3 主要内容为使用修改的代码在实际的开发板上进行，受实验条件的限制，本次实验必须以仿真方式进行。具体任务为修改程序测试指令的组成，修改前两句代码为 and 和 sra，重新运行程序，直到 r16 寄存器中的结果不再改变为止。

二． 实验过程

1.使用 vs code 编辑 lab.s，修改为题目要求。

```
1  .include "nios_macros.s"
2  .text
3  .equ TEST_NUM,0X90abcdef
4
5  .global _start
6  _start:
7      and r3,r7,r16
8      sra r7,r7,r3
9      mov r4,r7
10     STRING_COUNTER:
11     mov r2,r0
12     STRING_COUNTER_LOOP:
13     beq r4,r0,END_STRING_COUNTER
14     srli r5,r4,0x01
15     and r4,r4,r5
16     addi r2,r2,0x0001
17     br STRING_COUNTER_LOOP
18
19     END_STRING_COUNTER:
20     mov r16,r2
21
22     END:
23     br END
24 .end
```

2.重载 lab2 的程序保持寄存器不变，运行以上修改的程序数次：

The screenshots show the following states:

- Top Left:** Assembly view showing instructions from address 00000000 to 00000015. The register state shows R0-R7, STATUS, ESTATUS, and CPUID.
- Top Right:** Assembly view showing instructions from address 00000024 to 0000005C. The register state shows R0-R7, STATUS, ESTATUS, and CPUID.
- Bottom Left:** Assembly view showing instructions from address 00000024 to 0000005C. The register state shows R0-R7, STATUS, ESTATUS, and CPUID.
- Bottom Right:** Assembly view showing instructions from address 00000024 to 0000005C. The register state shows R0-R7, STATUS, ESTATUS, and CPUID.

	r2	r3	r7	r16
第一次	5	4	F90ABCDE	5
第二次	9	4	FF90ABCD	9
第三次	12	9	FFFFC855	12
第四次	20	10	FFFFFFFF	20

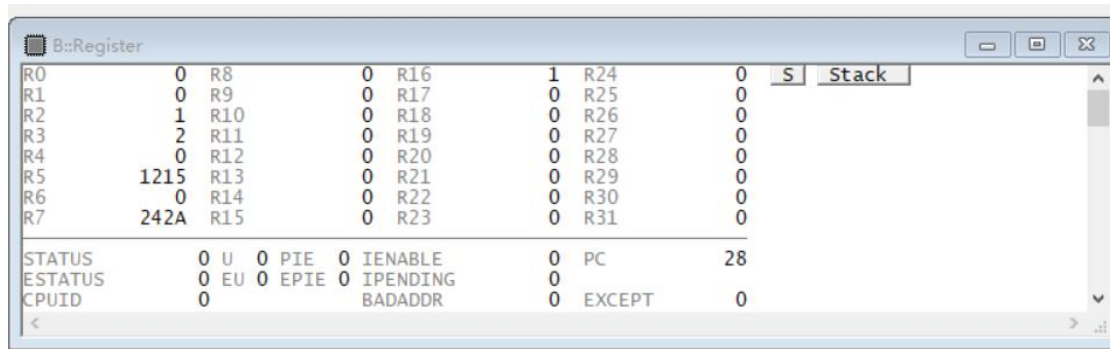
3.将 sra 改为 srli 再次运行。

B::Register									
R0	0	R8	0	R16	4	R24	0	S	Stack
R1	0	R9	0	R17	0	R25	0		
R2	4	R10	0	R18	0	R26	0		
R3	4	R11	0	R19	0	R27	0		
R4	0	R12	0	R20	0	R28	0		
R5	0201	R13	0	R21	0	R29	0		
R6	0	R14	0	R22	0	R30	0		
R7	090ABCDE	R15	0	R23	0	R31	0		
<hr/>									
STATUS	0	U	0	PIE	0	IENABLE	0	PC	28
ESTATUS	0	EU	0	EPIE	0	IPENDING	0		
CPUID	0			BADADDR		EXCEPT	0		

B::Register									
R0	0	R8	0	R16	4	R24	0	S	Stack
R1	0	R9	0	R17	0	R25	0		
R2	4	R10	0	R18	0	R26	0		
R3	4	R11	0	R19	0	R27	0		
R4	0	R12	0	R20	0	R28	0		
R5	20	R13	0	R21	0	R29	0		
R6	0	R14	0	R22	0	R30	0		
R7	0090ABCD	R15	0	R23	0	R31	0		
<hr/>									
STATUS	0	U	0	PIE	0	IENABLE	0	PC	28
ESTATUS	0	EU	0	EPIE	0	IPENDING	0		
CPUID	0			BADADDR		EXCEPT	0		

B::Register									
R0	0	R8	0	R16	4	R24	0	S	Stack
R1	0	R9	0	R17	0	R25	0		
R2	4	R10	0	R18	0	R26	0		
R3	4	R11	0	R19	0	R27	0		
R4	0	R12	0	R20	0	R28	0		
R5	2	R13	0	R21	0	R29	0		
R6	0	R14	0	R22	0	R30	0		
R7	00090ABC	R15	0	R23	0	R31	0		
<hr/>									
STATUS	0	U	0	PIE	0	IENABLE	0	PC	28
ESTATUS	0	EU	0	EPIE	0	IPENDING	0		
CPUID	0			BADADDR		EXCEPT	0		

B::Register									
R0	0	R8	0	R16	2	R24	0	S	Stack
R1	0	R9	0	R17	0	R25	0		
R2	2	R10	0	R18	0	R26	0		
R3	4	R11	0	R19	0	R27	0		
R4	0	R12	0	R20	0	R28	0		
R5	0	R13	0	R21	0	R29	0		
R6	0	R14	0	R22	0	R30	0		
R7	90AB	R15	0	R23	0	R31	0		
<hr/>									
STATUS	0	U	0	PIE	0	IENABLE	0	PC	28
ESTATUS	0	EU	0	EPIE	0	IPENDING	0		
CPUID	0			BADADDR		EXCEPT	0		



得到：

	R2	R3	R7	R16
第一次	4	4	090ABCDE	4
第二次	4	4	0090ABCD	4
第三次	4	4	00090ABC	4
第四次	2	4	90AB	2
第五次	1	2	242A	1
第六次	1	0	242A	1

三．代码分析

代码分析这里参考朱俊峰同学的解释：STRING_COUNTER 后的程序用于检测数据中连续的 1 的个数，修改代码时并没有改动这部分内容，改变的只是输入的数据，在原始测试数据的基础上，我们每次执行时都将测试数据右移或左移，移动的位数由移动前 r16 的结果和测试数据按位与得到，sra 右移时，高位由 1 补，结果导致连续的 1 增多，直到数据全部右移剩下 FFFFFFFF，r16 最终结果 20 即 32 个连续的 1。类似的，srl 右移时，高位由 0 补，连续的 1 减少，最后 r3 为 0，不再右移。所以会出现以上结果。

```

1  .include "nios_macros.s"
2  .text
3  .equ TEST_NUM,0x90abcdef
4
5  .global_start
6  start:
7      and r3,r7,r16
8      sra r7,r7,r3
9      mov r4,r7
10     STRING_COUNTER:
11     mov r2,r0
12     STRING_COUNTER_LOOP:
13     beq r4,r0,END_STRING_COUNTER
14     srli r5,r4,0x01
15     and r4,r4,r5
16     addi r2,r2,0x0001
17     br STRING_COUNTER_LOOP
18
19     END_STRING_COUNTER:
20     mov r16,r2
21
22     END:
23     br END
24 .end

```

四．组长点评

本实验的目的是了解和分析指令的构成。指令是放置在 ROM 中的特殊二进制数。再仿真中的 PC，这个寄存器代表当前程序运行的位置。

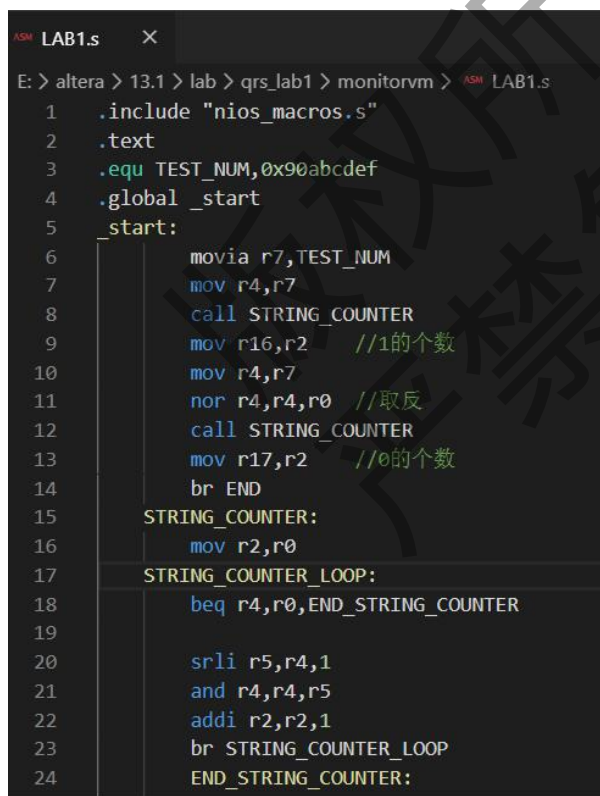
第四部分 Part4/Lab1

一． 题目分析

本实验目的为实验子程序的功能，并考验对汇编语言掌握程度。本实验代码可以由 c 语言反推。在 C 语言中，可以使用一个带参函数实现特定功能，这个函数就是此次汇编中的子函数，主要是计数，即可。此后两次调用子程序，只要入口参数不一样即可。

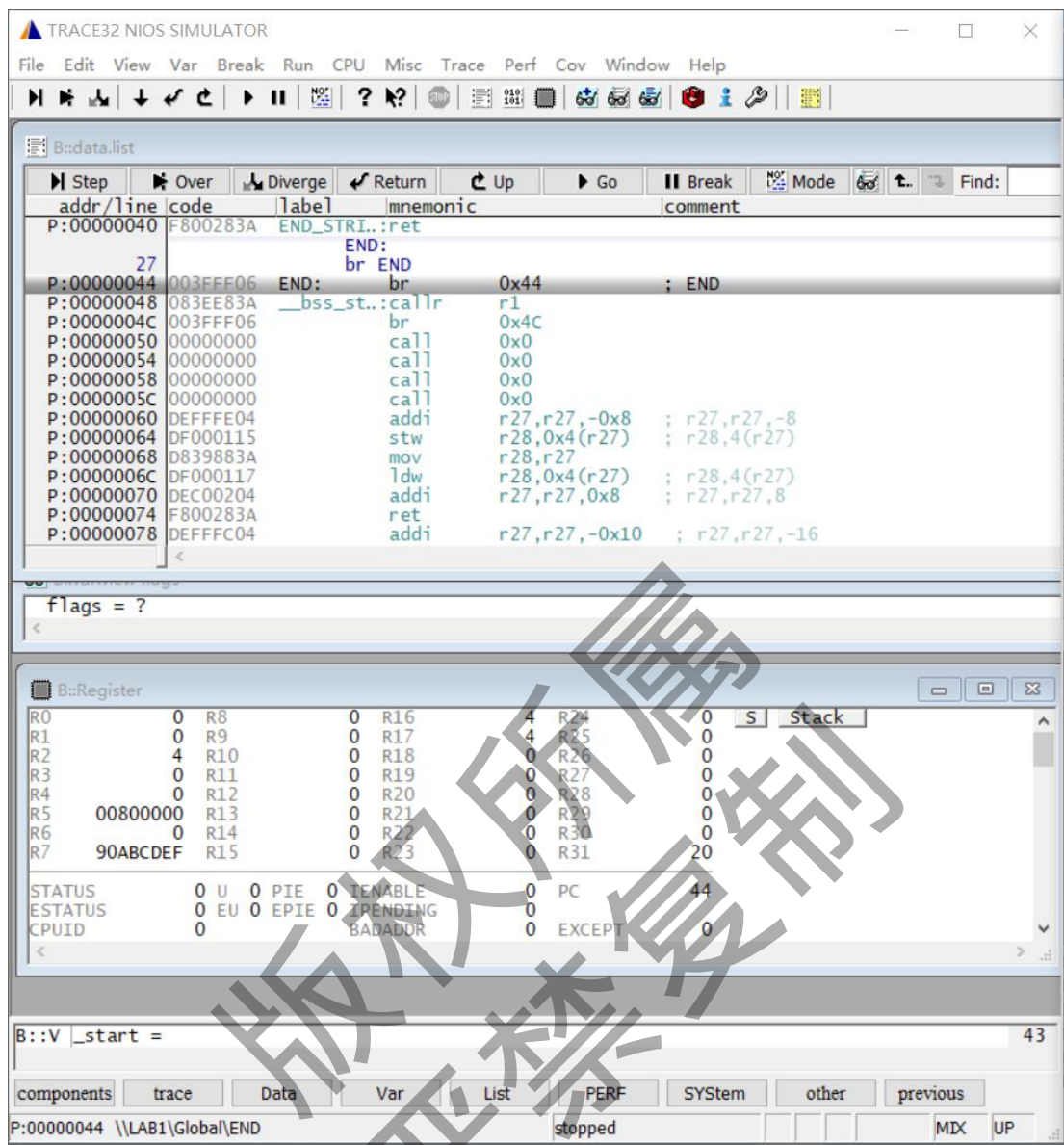
二． 实验过程

1.使用 vs code 编辑 lab.s，修改为题目要求。



```
ASM LAB1.s x
E: > altera > 13.1 > lab > qrs_lab1 > monitorvm > ASM LAB1.s
1  .include "nios_macros.s"
2  .text
3  .equ TEST_NUM,0x90abcdef
4  .global _start
5  _start:
6      movia r7,TEST_NUM
7      mov r4,r7
8      call STRING_COUNTER
9      mov r16,r2    //1的个数
10     mov r4,r7
11     nor r4,r4,r0  //取反
12     call STRING_COUNTER
13     mov r17,r2    //0的个数
14     br END
15     STRING_COUNTER:
16         mov r2,r0
17     STRING_COUNTER_LOOP:
18         beq r4,r0,END_STRING_COUNTER
19
20         srli r5,r4,1
21         and r4,r4,r5
22         addi r2,r2,1
23         br STRING_COUNTER_LOOP
24     END_STRING_COUNTER:
```

2.重复编译仿真步骤，观察 R16 和 R17 的值，均为 4.



三．代码分析

代码主体跟实验二基本一样，主要改动为，将计数部分单独拿出，并作为子程序。另外将 `TEXT_NUM` 取反之后，再经过计数部分计数。即可得到想要的程序。


```
ASM LAB1.s X
E: > altera > 13.1 > lab > qrs_lab1 > monitorvm > ASM LAB1.s
1 .include "nios_macros.s"
2 .text
3 .equ TEST_NUM,0x90abcdef
4 .global _start
5 _start:
6     movia r7,TEST_NUM
7     mov r4,r7
8     call STRING_COUNTER
9     mov r16,r2    //1的个数
10    mov r4,r7
11    nor r4,r4,r0  //取反
12    call STRING_COUNTER
13    mov r17,r2    //0的个数
14    br END
15    STRING_COUNTER:
16        mov r2,r0
17    STRING_COUNTER_LOOP:
18        beq r4,r0,END_STRING_COUNTER
19
20        srl r5,r4,1
21        and r4,r4,r5
22        addi r2,r2,1
23        br STRING_COUNTER_LOOP
24    END_STRING_COUNTER:
```

四 . 组长点评

本实验的难点在于子程序的编写和调用，取反操作，及 **end** 的位置等问题。可以使用间接取反等操作来进行修改。完成度比较高。

第四部分 Part4/Lab1

一． 题目分析

本实验参考借鉴朱俊峰同学的案例，使用上一个实验的子程序对数据中交替的 1 和 0 计数，而题目中题型将数据左移一位或右移一位与原数据异或。发现连续的 01 交替在经过此操作之后会变成连续的 1。

此时再此调用 part2 中的操作即可解决问题。

二． 实验过程

1. 使用 vs code 编辑 lab.s，修改为题目要求。

```
1 .include "nios_macros.s"
2 .text
3 .equ TEST_NUM,0x90abcdef
4
5 .global_start
6 start:
7     movia r7,TEST_NUM
8     mov r4,r7
9     call STRING_COUNTER
10    br end
11    STRING_COUNTER:
12    mov r2,r0
13    slli r5,r4,0x01
14    xor r4,r4,r5
15    STRING_COUNTER_LOOP:
16    beq r4,r0,END_STRING_COUNTER
17    srli r5,r4,1
18    and r4,r4,r5
19    addi r2,r2,1
20    br STRING_COUNTER_LOOP
21    END_STRING_COUNTER:
22    addi r2,r2,1
23    mov r18,r2
24    ret
25    END:|
26    br END
27 .end
```

- 2.重复编译仿真步骤，观察 R2 和 R18 的值，均为 8，实验成功。

B::Register

R0	0	R8	0	R16	0	R24	0	S	Stack
R1	0	R9	0	R17	0	R25	0		
R2	8	R10	0	R18	8	R26	0		
R3	0	R11	0	R19	0	R27	0		
R4	0	R12	0	R20	0	R28	0		
R5	00020000	R13	0	R21	0	R29	0		
R6	0	R14	0	R22	0	R30	0		
R7	90ABCDEF	R15	0	R23	0	R31	10		
STATUS	0	U	0	PIE	0	IENABLE	0	PC	40
ESTATUS	0	EU	0	EPIE	0	IPENDING	0		
CPUID	0			BADADDR	0	EXCEPT	0		

三 . 代码分析

代码参考了朱俊峰同学，主要使用到 slli, xor 两条指令。符合题目中的帮助，即左移一位与自己异或。这两句添加到循环体之前，其它保持不变。

```

1 .include "nios_macros.s"
2 .text
3 .equ TEST_NUM,0x90abcdef
4
5 .global _start
6 _start:
7     movia r7,TEST_NUM
8     mov r4,r7
9     call STRING_COUNTER
10    br end
11    STRING_COUNTER:
12    mov r2,r0
13    slli r5,r4,0x01
14    xor r4,r4,r5
15    STRING_COUNTER_LOOP:
16    beq r4,r0,END_STRING_COUNTER
17    srli r5,r4,1
18    and r4,r4,r5
19    addi r2,r2,1
20    br STRING_COUNTER_LOOP
21    END_STRING_COUNTER:
22    addi r2,r2,1
23    mov r18,r2
24    ret
25    END:|
26    br END
27 .end

```

四 . 组长点评

本次实验左移异或加 1 是需要我们进行数学演算的一件事情。本题在题目指导下，完成度比较高。

第六部分 Part 6/Lab1

一、题目分析

本实验要求我们使用 c 语言编写一个同样功能的函数，计数数据中连续的 1 个数，编译后观察其汇编语言与之前编写的有何不同。这里我们使用 Quartus 自带的 NiosII SBT for Eclipse 编写 c 程序，然后将生成的 elf 文件导入到 simnios，观察编译器。

二、实验过程

1. 打开 Quartus, 可以按照正点原子的步骤在 Tools 菜单下点击 NiosII SBT for Eclipse。新建工程，选择之前生成的 Nios 系统，导入 sopcinfo 和 sof 文件，以 hello world 为模板生成 BSP 程序，在系统自动生成的 hello world.c 上编写测试程序。

```
15
16
17 #include <stdio.h>
18
19
20 unsigned int num = 0x90abcdef,r4,r5,r2
21 int main()
22 {
23     r4=num;
24     while(r4)
25     {
26         r5=r4>>1;
27         r4=r4&r5;
28         r2++;
29     }
30     return 0;
31 }
32
```

2. 按照正点原子的步骤取消使用 c++ 以及其它冗余的设定，编译工程，编译通过后可以看到生成了 elf 文件。

hal

sys_clk_timer:

none

timestamp_timer:

none

stdin:

jtag_uart_0

stdout:

jtag_uart_0

stderr:

jtag_uart_0

☒ enable_small_c_library

☐ enable_gprof

☒ enable_reduced_device_drivers

☐ enable_sim_optimize

hal

max_file_descriptors:

32

☒ enable_instruction_related_exceptions_api

log_port:

none

☒ enable_exit

☐ enable_clean_exit

☐ enable_runtime_stack_checking

☐ enable_c_plus_plus

☐ enable_lightweight_device_driver_api

☐ enable_mul_div_emulation

☒ enable_sopc_sysid_check

3. 打开 simnios，导入 elf 文件，清空 CPU，在反汇编中找到 main 函数，然后设置断点。

B::data.list

Step Over Diverge Return Up Go Break Mode

addr/line	code	label	mnemonic	comment
P:00008088	1004D07A		srl r2,r2,0x1	; r2,r2,1
P:0000808C	D0A08515		stw r2,-0x7DEC(r26)	; r2,-32236(r26)
		r4=r4&r5;		
P:00008090	D0E08417		ldw r3,-0x7DF0(r26)	; r3,-32240(r26)
P:00008094	D0A08517		ldw r2,-0x7DEC(r26)	; r2,-32236(r26)
P:00008098	1884703A		and r2,r3,r2	
P:0000809C	D0A08415		stw r2,-0x7DF0(r26)	; r2,-32240(r26)
		r2++;		
P:000080A0	D0A08317		ldw r2,-0x7DF4(r26)	; r2,-32244(r26)
P:000080A4	10800044		add r2,r2,0x1	; r2,r2,1
P:000080A8	D0A08315		stw r2,-0x7DF4(r26)	; r2,-32244(r26)
		while(r4)		
P:000080AC	D0A08417		ldw r2,-0x7DF0(r26)	; r2,-32240(r26)
P:000080B0	103FF41E		bne r2,r0,0x8084	
		}		
30		return 0;		
P:000080B4	0005883A		mov r2,r0	

B::var.view flags

flags = ?

B::Register.view

R0	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	R16	R17	R18	R19	R20	R21	R22	R23	R24	R25	R26	R27	R28	R29	R30	R31
0	05A0	4	4021	0	0	0	FFFFFFF	DEADBEEF	DEADBEEF	DEADBEEF	DEADBEEF	DEADBEEF	DEADBEEF	DEADBEEF	DEADBEEF	DEADBEEF	DEADBEEF	DEADBEEF	DEADBEEF	DEADBEEF	DEADBEEF	DEADBEEF	DEADBEEF	DEADBEEF	DEADBEEF	DEADBEEF	DEADBEEF	DEADBEEF	DEADBEEF	DEADBEEF	DEADBEEF

S Stack

STATUS	ESTATUS	CPUID	PC	EXCEPT
1 U 0 PIE 1 IENABLE	0 EU 0 EPIE 0 IPENDING	0	80A8	0

4. 运行程序，一步一步观察寄存器 R2 的变化，看到从 0 加到 4，看到 r2 的值是 4，说明程序的功能是正确的。

三、代码详解

C 语言比较显而易见，就是将汇编语言简单化。除此之外，使用 C 语言生成的 elf 中出现很多初始化的步骤，这也是 C 语言比较冗余的地方。查看同学开源后得知，r28 其实是 Nios 中具有特殊功能的帧指针，调用 r28，程序将变量暂存到内存中，计算时再将其读出，结束后刷新内存中的值，通用寄存器只用于工作而不用与保存结果。

四、组长点评

C 语言与汇编语言的差距，包括异同等问题做了探讨，主要在于 C 语言会自动寻址，而这个寻址过程正是汇编没有的。除此之外，还调用一个寄存器暂存变量到内存中。这一点也有所不同。

版权所有
禁止转载