**组号： 2**

****



计算机系统综合课程设计

（OS课题）作业报告

组长：09019xxxXXX

成员：09019xxxXXX

09019xxxXXX

09019xxxXXX

09019xxxXXX

东南大学计算机学院

二〇二二年 11 月

**目录**

[问题1 2](#_Toc120362825)

[问题2 5](#_Toc120362826)

[问题3 6](#_Toc120362827)

[问题4 8](#_Toc120362828)

[问题5 12](#_Toc120362829)

[问题6 13](#_Toc120362830)

[问题7 14](#_Toc120362831)

# 问题1

Minisys汇编程序设计4：综合应用（MOOC16讲的作业4）

程序内部有一16位变量VAL，YLD7~0~GLD7~0始终输出VAL的值。

SW23/SW22/SW21为功能选择，含义如下表：

|  |  |  |  |
| --- | --- | --- | --- |
| SW23 | SW22 | SW21 | 动作 |
| X | 0 | 0 | 无动作 |
| 0 | 0 | 1 | 将SW15~SW0这16位作为输入赋值给VAL |
| 0 | 1 | 0 | VAL=VAL+1 （每隔约1秒动作一次） |
| 0 | 1 | 1 | VAL=VAL-1（每隔约1秒动作一次） |
| 1 | 0 | 1 | VAL左移1位（每隔约1秒动作一次） |
| 1 | 1 | 0 | VAL逻辑右移1位（每隔约1秒动作一次） |
| 1 | 1 | 1 | VAL算术右移1位（每隔约1秒动作一次） |

**答：具体代码见下。**

1. LEAF(q1\_code)
2. \_start\_pos:
3. *# 赋值 YLD[7:0], GLD[7:0] 为 VAL*
4. ins     YLD, VAL, 8, 8
5. ins     GLD, VAL, 0, 8
6. *# 判断 SW[23:21]*
7. ins     t3, SW, 23, 1
8. ins     t2, SW, 22, 1
9. ins     t1, SW, 21, 1
10. *# 如果 SW[22] = SW[21] = 0*
11. or      t0, t2, t1
12. beqz    t0, \_sw\_22\_0\_21\_0
13. *# 如果 0 0 1*
14. or      t0, t3, t2
15. not     t0, t0
16. and     t0, t0, t1
17. bnez    t0, \_sw\_23\_0\_22\_0\_21\_1
18. *# 其他情况：接下来我们先耗费掉 1s*
19. *# 假设 CPU 1s 执行 20M 条指令*
20. li      t8, 0x7 0000 0000 *# 7M*
21. \_repeat\_to\_add:
22. addit8, t8, -0x1
23. beq     t8, zero, \_lets\_go\_on
24. j       \_repeat\_to\_add
25. \_lets\_go\_on:
26. *# 现在来判断 SW[23:21]*
27. beqz    t3, \_sw\_23\_0
28. j       \_sw\_23\_1
29. \_sw\_23\_0:
30. beqz    t1, \_sw\_23\_0\_22\_1\_21\_1
31. bnez    t1, \_sw\_23\_0\_22\_1\_21\_0
32. j       \_end\_pos
33. \_sw\_23\_1:
34. and     t0, t2, t1 *# 22 21 都为 1 时，t0 = 1*
35. bnez    t0, \_sw\_23\_1\_22\_1\_21\_1
36. beqz    t1, \_sw\_23\_1\_22\_0\_21\_1
37. beqz    t2, \_sw\_23\_1\_22\_1\_21\_0
38. j       \_end\_pos
40. \_sw\_22\_0\_21\_0: *# 无动作*
41. j       \_end\_pos
42. \_sw\_23\_0\_22\_0\_21\_1: *# 赋值 VAL <- SW[15:0]*
43. ins     VAL, SW, 0, 16
44. j       \_end\_pos
45. \_sw\_23\_0\_22\_1\_21\_0: *# VAL += 1，1s 一次*
46. addi    VAL, VAL, 0x1
47. j       \_end\_pos
48. \_sw\_23\_0\_22\_1\_21\_1: *# VAL -= 1，1s 一次*
49. addi    VAL, VAL, -0x1
50. j       \_end\_pos
51. \_sw\_23\_1\_22\_0\_21\_1: *# VAL <<= 1，1s 一次*
52. sll    VAL, VAL, 0x1
53. j       \_end\_pos
54. \_sw\_23\_1\_22\_1\_21\_0: *# VAL >>= 1 逻辑右移，1s 一次*
55. srl    VAL, VAL, 0x1
56. j       \_end\_pos
57. \_sw\_23\_1\_22\_1\_21\_1: *# VAL >>= 1 算术右移，1s 一次*
58. srv    VAL, VAL, 0x1
59. j       \_end\_pos
60. \_end\_pos:
61. j       \_start\_pos
62. END(q1\_code)

# 问题2

要在MIPSFpga上开发操作系统，首先需要对导入的MIPSfpga Core进行硬件上的扩充，这主要表现在哪些方面，请简要说明各个方面的主要作用？

**答：具体如下表所示。**

|  |  |
| --- | --- |
| 时钟 | 为MIPS CPU配置合适频率的时钟信号。 |
| 总线转换 | 为了方便直接使用Vivado提供的IP核，这里需要将MIPS软核的AHB-Lite 总线，转换成常用的AXI总线标准。 |
| GPIO设备 | 为LED、蜂鸣器、七段数码管、拨码开关等外部设备，提供控制模块。 |
| 存储系统 | 包括Block RAM（cache）、DDR3 SDRAM（内存）等，我们要为其提供控制模块。 |
| 其他外设 | 提供UART、定时器、SD卡等的控制设备。 |
| 外部引脚 | 即与板上的引脚对应的Input Output信号。 |

# 问题3

1. 请简述MIPSfpga上操作系统的启动过程。

**答：**

上电后，mips core 首先从虚拟地址的 0xbfc0 0000（固定映射到物理地址空间的 0x1fc0 0000）处开始执行 bootloader 代码。bootloader 包含了两部分内容：boot（对硬件进行检测、初始化）和 load（加载 OS）， 执行完之后进入 OS 内核。

具体的，bootloader 部分执行了以下工作：

* **初始化 CP0 寄存器组**：包括对 status、cause、config 等寄存器的初始化。
* **初始化 TLB**：初始化 TLB 时，如果两个 entry 对应同一个虚拟地址，可能会出现 “TLB Shutdown” 错误。所以，要给每个 entry 赋值不同的虚拟地址，建议以 8K 作为间隔。
* **初始化 cache**：MIPS 的 cache 分为 instruction cache 与 data cache，初始化时，要把所有 cache tag 置为 invalid（通过修改 tag 为全零）。达成“初始化所有 cache”的方法，是 1. 首先读 CP0 config 寄存器，得到 icache dcache 的总数，2. 维护一个计数器，将 cache 一个一个初始化。
* **加载 OS 内核**：首先，通过文件系统访问磁盘，通过loader程序将内核 elf 文件被读进一段内存缓冲区内。然后，我们解析 elf 文件的头标，得到 elf 程序段表的位置；解析 elf 文件的程序段表，按照段表的信息，将每个程序段加载到相应的地址中。

接下来我们进入 OS 内核，进行 OS 的初始化，初始化外设、内存、页表、进程管理、异常向量，之后开启 shell 进程。具体的，

* 对于外部设备的初始化，调用驱动的初始化方法，将外设内容置零。
* 在内存管理部分，首先将内核代码段、数据段映射到 0x8000 0000（内核虚拟地址），然后将 0x8040 0000 处作为内核栈的栈底。函数 vm\_init() 通过调用 alloc() 给页表（pgdir）、内存控制块数组（pages）和进程控制块数组（envs）分配所需的物理内存。page\_init() 将未分配的物理页加入到空闲链表 page\_free\_list 中去。至此，页表构建完成，可以使用page\_alloc() 以页为单位请求分配内存了。
* 对于异常向量的初始化，将各异常处理函数的地址赋给中断向量表。
* 在进程管理部分，首先在内存中开辟出固定数量的进程管理块 PCB，并都链接至 env\_free\_list 链表上，方便接下来新建进程的管理。
* 最后，调用进程管理，新建 shell 进程。

1. MIPSfpga中断使能的前提条件是什么？分别涉及到几个什么寄存器？该前提条件说明中断和调试状态是什么关系？

**答：**

* **MIPSfpga 中断使能的前提条件**

中断是程序正常执行过程中，由于外部某些条件所触发的需要在正常程序流以外处理的情况，并且在异常处理的过程中需要进入内核态，由操作系统进行上下文的保存恢复和执行处理程序。因此在某段程序执行的过程中使能中断的一个重要前提是该程序允许被打断（或者打断其执行并不影响其后正常的执行），一般来说多在用户态中使能中断，而在内核态中，有时在一些核心处理程序中，为了避免嵌套异常和中断的发生，常选择关闭中断以保证其可以一次性执行完成。

* **涉及到几个什么寄存器**

在MIPSfpga中，中断是否使能的相关状态保存在 CP0\_Status 寄存器中，通过其IE位来表示，当该寄存器中 IE = 1，EXL = 0，ERL = 0 时，中断才会被启动。 在 OS 代码中，开关中断的控制代码位于 inc/stackframe.S 中：

1. *# 开中断*
2. .macro STI
3. mfc0 t0, CP0\_STATUS
4. li t1, (STATUS\_CU0 | 0x1)
5. or t0, t1  *# or rd, rs, rt => GPR[rd] <- GPR[rs] OR GPR[rt]*
6. mtc0 t0, CP0\_STATUS
8. .endm
9. *# 关中断*
10. .macro CLI
11. mfc0 t0, CP0\_STATUS
12. li t1, (STATUS\_CU0 | 0x1)
13. or t0, t1
14. xor t0, 0x1
15. mtc0 t0, CP0\_STATUS
16. .endm

可以看出，OS 通过操作 CP0\_Status 中的若干位实现了中断使能的相关操作。

* **该前提条件说明中断和调试状态是什么关系**

调试状态中的断点其实也是一种中断，需要破坏程序的正常执行状态，在断点处触发中断以让程序停顿。因此在调试状态下，一定要启动中断使能，不然无法陷入异常而实现程序的单步执行操作。

# 问题4

请画出常规异常（非TLB缺失）的软件处理流程（可给出适当解释）。

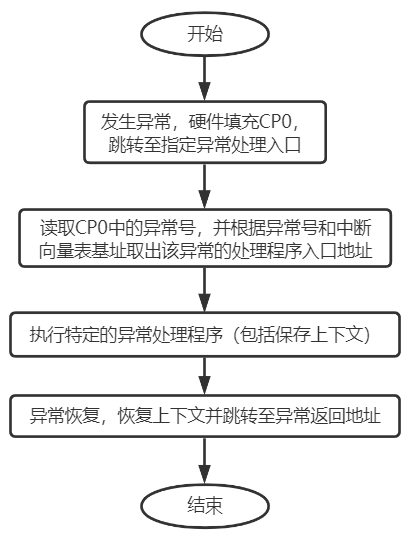
**答：**

* **异常总体处理流程**

异常处理的总体逻辑可以归纳为如下步骤：

1. 硬件触发中断，触发的时候由硬件填充CP0中相关的寄存器（如EPC寄存器用来保存当前PC值），并且保持其余寄存器的值不改变，跳转至异常处理程序的固定入口地址（本OS中为0x80000000）。
2. 进入异常处理程序之后，首先根据CP0中填充的异常号，根据异常向量表基址+异常号偏移，找到相应异常的特定处理程序地址，然后跳转过去进行执行。值得注意的是，由于各异常处理程序对上下文保存的要求不同（例如时钟到期异常并不需要向内核栈中保存上下文），因此上下文的保存在各特定处理程序的开头再执行。
3. 在特定异常处理程序执行完成后，进入异常恢复程序，首先需要从内核栈中恢复上下文，然后从EPC中读出异常返回地址，跳转至该地址继续执行即可。

异常总体处理的流程图如下：

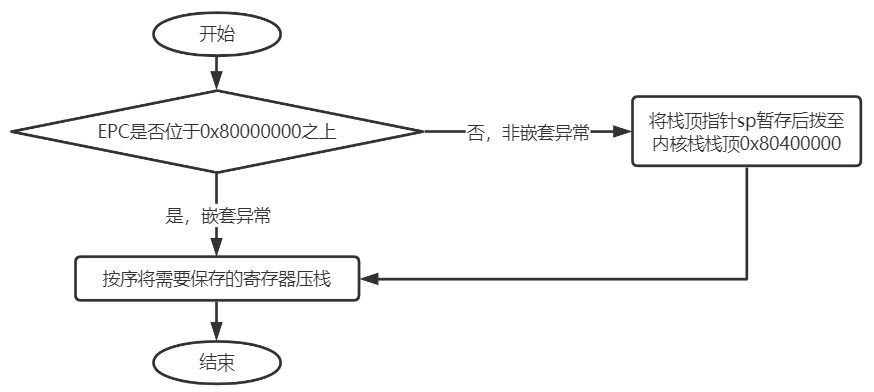


* **上下文保存处理**

上下文保存是跳转至特定异常处理程序后的第一步，对于一般的将上下文保存至内核栈的处理中，主要的步骤如下：

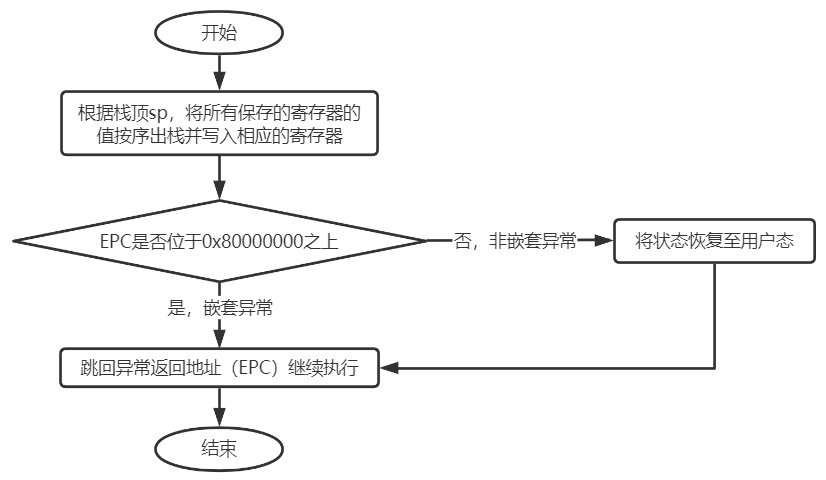
1. 通过EPC（异常返回地址）是否位于0x80000000之上来判断此次异常是否是嵌套异常，对于第一次发生的异常（非嵌套异常），需要首先将其栈顶指针sp暂存后拨到内核栈顶0x80400000，而对于嵌套异常，sp已经在内核栈中，因此在此基础上继续压栈即可。
2. 确定栈顶指针sp的位置之后，只需要依次将需要进行保存的寄存器的值压入栈中即可。

上下文保存处理的流程图如下：



* **异常恢复处理**

异常处理结束后，需要恢复上下文并跳转回发生异常的地址继续执行，在该过程中，仍需要判断此次异常是否嵌套，从而决定异常结束后是否需要跳回用户态。异常恢复处理的流程图如下：

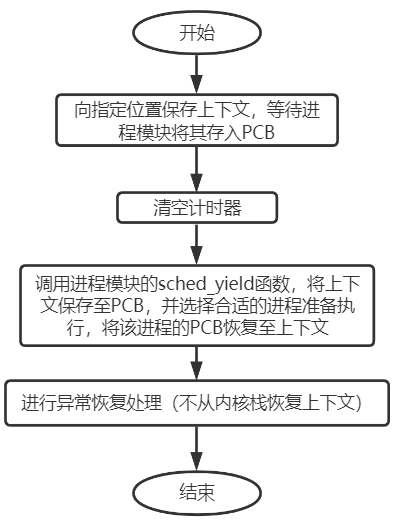


* **时钟中断异常处理**

时钟中断异常说明当前时间片已经耗尽，接下来需要切换进程，因此主要的处理过程如下：

1. 保存上下文。由于时钟中断异常会导致进程切换，因此并不需要向内核栈保存上下文，而是需要将上下文存至指定位置，方便之后的进程处理模块将其保存至PCB中。
2. 清空计时器。
3. 指定调度函数，选择合适的进程作为下一个执行进程，并获得其起始地址。
4. 进行异常恢复处理，但是不从内核栈恢复上下文。

时钟中断异常处理的流程图如下：



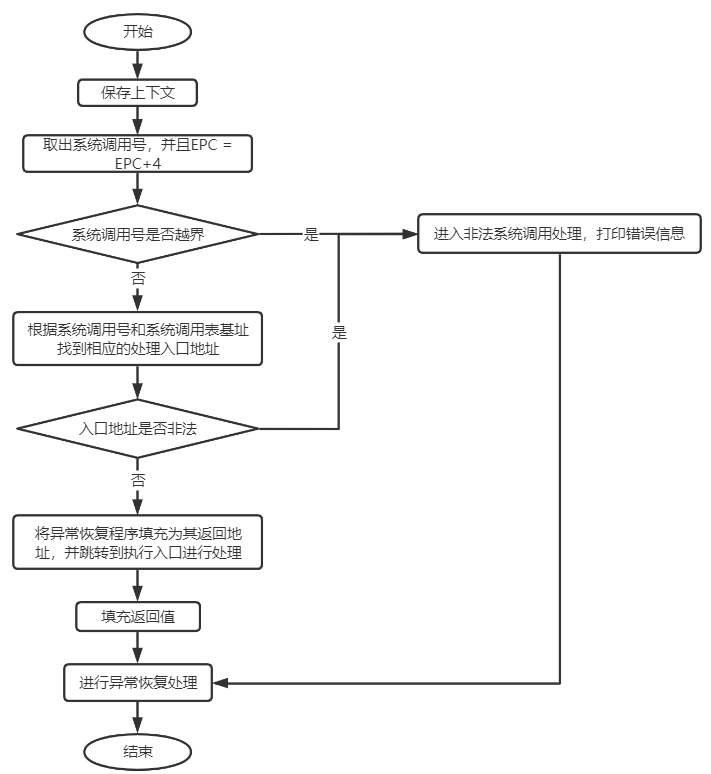
* **系统调用异常处理**

在用户进行系统调用时，操作系统陷入异常并进入内核态，根据用户的调用号找到相应的程序执行系统调用，主要的处理过程如下：

1. 保存上下文。
2. 首先取出系统调用号（作为参数从a0传入）。
3. 将EPC+4（因为系统调用需要返回的是下一条指令而非当前指令），然后判断系统调用号是否越界，如果越界，跳转至非法系统调用处理程序，打印错误信息并直接进行异常恢复处理。
4. 对于正确的系统调用，根据系统调用向量表基址+调用号偏移，找到该系统调用的处理入口地址，并且判断该入口地址是否合法，对于不合法的入口地址，同样跳转至非法系统调用处理程序。
5. 对于合法的系统调用，将异常恢复程序填充为其返回地址后，跳转到执行入口进行处理。
6. 处理结束后进行异常恢复处理，保存返回值，恢复上下文。

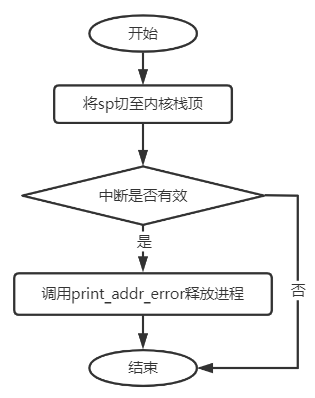
系统调用异常处理的流程图如下：

(见下页)



* **地址越界异常处理**

当用户访问到内核地址空间之后，会自动触发地址越界异常，该异常的处理比较简单，不需要上下文的保存和恢复。对于触发该异常的用户进程，会直接被释放掉，因此在进程创建时，经常将最终的返回地址至于内核地址空间，这样其在返回时就会触发地址越界异常并被释放。地址越界异常处理的流程图如下：



# 问题5

在我们设计的OS中，有哪些中断和异常要处理？这些中断和异常的处理函数是什么？处理返回的特点是什么？请填写下表。

|  |  |  |
| --- | --- | --- |
| 中断或异常名 | 处理函数 | 返回时的特点 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**答：具体如下表所示。**

|  |  |  |
| --- | --- | --- |
| 中断或异常名 | 处理函数 | 返回时的特点 |
| 时钟中断 | handle\_int | 不需要从内核栈中恢复上下文 |
| 系统调用 | handle\_sys | 需要将系统调用的返回值保存后再返回 |
| 地址访问越界 | handle\_addr | 进程被释放，无需返回 |
| TLB异常 | handle\_tlb | 正常返回，从内核栈恢复上下文 |

# 问题6

关于虚拟地址，请完善下表。

|  |  |  |
| --- | --- | --- |
| 名称 | 起始虚拟地址 | 物理地址（如果是确定的） |
| 上电起始地址 |  |  |
| 内核代码段 |  |  |
| 中断和异常 |  |  |
| 页目录基址 |  |  |
| 堆区 |  |  |
| 用户栈区 |  |  |
| 用户的代码段 |  |  |

**答：具体如下表所示。**

除了栈区为向下生长，表中其余区域为向上生长（向高地址方向增长）。

|  |  |  |
| --- | --- | --- |
| 名称 | 起始虚拟地址 | 物理地址（如果是确定的） |
| 上电起始地址 | 0xbfc0 0000 | 0x1fc0 0000 |
| 内核代码段 | 0x8001 0000 | 0x0001 0000 |
| 中断和异常 | 0xa000 0000 | 0x0000 0000 |
| 页目录基址 | 0x8040 0000 | 0x0040 0000 |
| 堆区 | 0x7f40 0000 | 创建进程时动态确定 |
| 用户栈区 | 0x7f3f e000（向下生长） | 创建进程时动态确定 |
| 用户的代码段 | 0x1500 0000 | 创建进程时动态确定 |

# 问题7

为什么在创建进程的时候要把进程返回地址设置成0x9000 0000？进程被调度有哪几种情况？起因是什么？

**答：**

* **关于进程返回地址 0x9000 0000**

根据内存空间布局设定，0x80000000 以上的地址空间为内核所使用的空间，当用户进程访问这些地址时，会自动触发地址访问越界异常，根据我们的实现，在中断处理程序 handle\_addr 中，将自动释放当前运行的进程 curenv。

因此，在进程初始化时，将每一个 env 的 ra（返回地址） 都设置为 0x90000000，这样进程运行结束时回返回 ra，从而触发 handle\_addr，实现自动释放运行结束的进程。

* **进程被调度的情况及起因**

在本课题所完成的操作系统中，进程被调度的情况可大致总结概括为如下几种：

1. 用完CPU时间片。
2. 正在执行的进程执行完毕。
3. 进程通过系统调用主动放弃CPU时间。
4. 中断、异常及系统调用以及返回到用户态时。

注：关于其他常见的引起进程调度的原因，如进程提出I/O请求后被阻塞等，因为在本操作系统中似乎只有忙等，故不列入。