

# 第十三章 自陷例程和中断

# 造成混乱的情况

- 如果允许应用程序员（用户程序员）直接访问KBDR和KBSR等来实现I/O的行为
  - I/O行为包含了被许多程序所共享的设备寄存器的使用
  - 用户程序员没有谨慎处理，给其他用户程序制造混乱

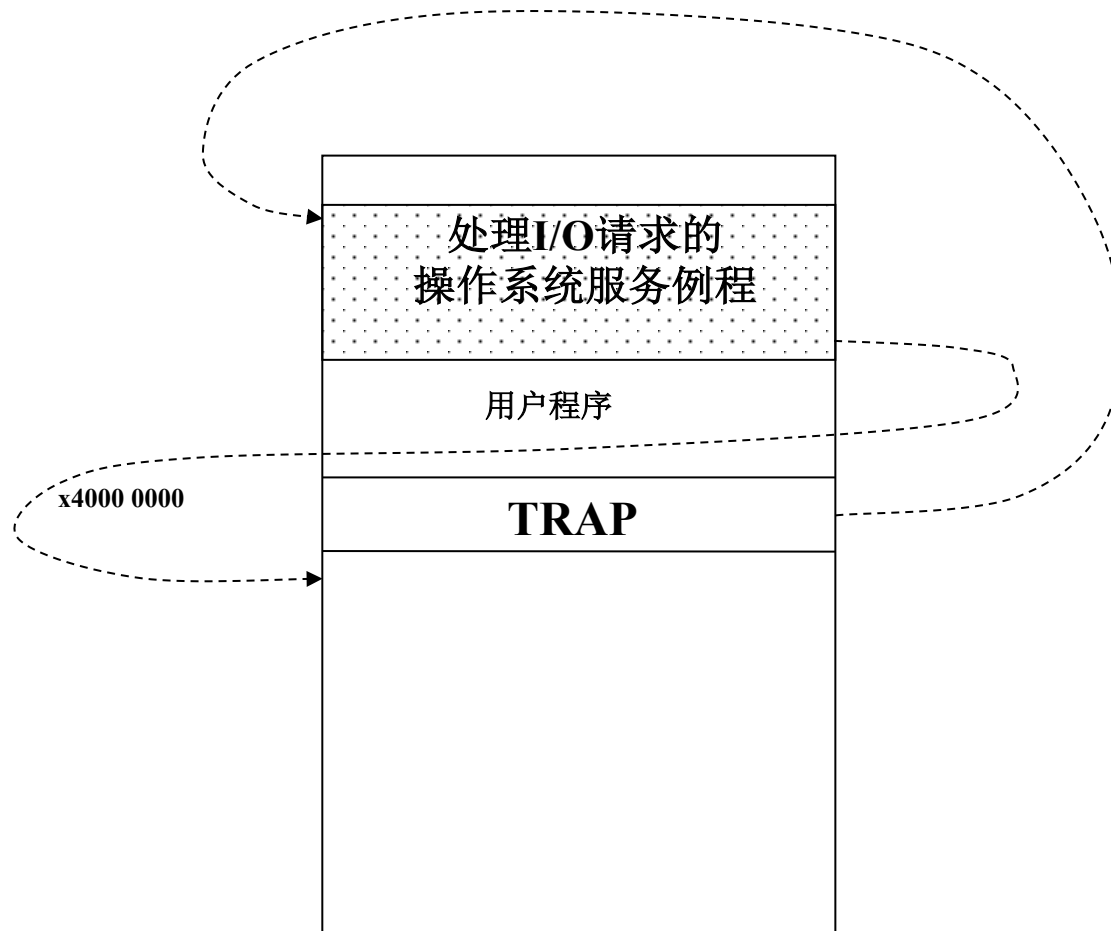
# 特权

- 硬件寄存器是有**特权**的
- 不拥有适当特权级别的程序不能访问

# 解决方案

- 自陷（TRAP）指令
- 操作系统
  - 拥有适当的特权级别
  - 用户程序员不需要在这个层面上理解I/O

# 系统调用



# TRAP机制-1

- **服务例程**

- 操作系统的一部分，代表用户程序执行的一组程序
- **DLX：256个服务例程**

TRAP向量	符号	作用
x06	GETC	从键盘读取一个字符，将其ASCII码复制到R4[7:0]
x07	OUT	将R4[7:0]中的字符输出到显示器
x08	PUTS	将R4所指的地址开头的一个字符串输出到显示器，每个字符占用一个存储单元，字符串以x00终止
x09	IN	输出一个提示符到显示器，从键盘读取一个字符，将其ASCII码复制到R4[7:0]，并将其回显到显示器上
x0A	GETS	需要2个参数： R4（字符串起始地址）和R5（长度n）。从键盘读取n-1个字符，如果输入小于n-1，则至回车结束，读入缓冲区中，并在后面加上一个x00。
x00	HALT	停止执行

# TRAP机制-2

- **TRAP向量表**

- 包括了256个服务例程的**起始地址的表**，每个起始地址需要占用4个连续的存储单元
  - 256个服务例程需要 $256 \times 4$ 个单元
  - 这张表被存储在存储单元的x0000 0000到x0000 03FF中
- 命名：**系统控制块**或TRAP向量表

x0000 0000-x0000 0003	x0000 0100
⋮	⋮
x0000 0018-x0000 001B	x0000 2500
x0000 001C-x0000 001F	x0000 2900
x0000 0020-x0000 0023	x0000 2D00
x0000 0024-x0000 0027	x0000 3100
x0000 0028-x0000 002B	x0000 3500
⋮	⋮

# 数据区/代码区

- 数据区的起始地址：代码区起始地址之前的x100个单元

x0000 0000-x0000 0003	xFFFFE 0100
⋮	⋮
x0000 0018-x0000 001B	x0000 2500
x0000 001C-x0000 001F	x0000 2900
x0000 0020-x0000 0023	x0000 2D00
x0000 0024-x0000 0027	x0000 3100
x0000 0028-x0000 002B	x0000 3500
⋮	⋮

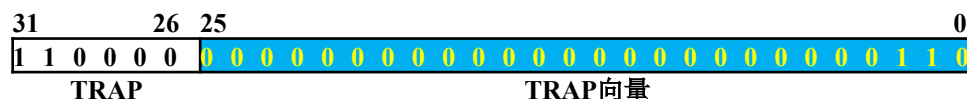


# TRAP机制-3

- **TRAP指令**

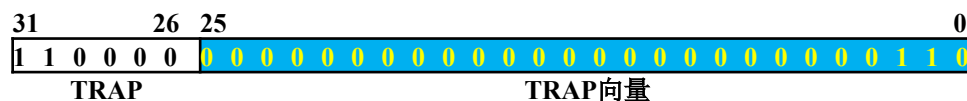
- 操作系统代表用户程序执行某一个服务例程，然后把控制权交还给用户程序
  - 根据TRAP向量， $PC \leftarrow$  相应服务例程首地址
  - 提供返回路径/“链接”

# TRAP指令（1~3阶段）



- 取指令；
  - $PC \leftarrow PC + 4$
- 译码；
- 计算有效地址：
  - TRAP向量（26位）扩展到32位，再**左移2位**（即乘以4）
    - $x06 \rightarrow x0000\ 0018$

# TRAP指令（4阶段）

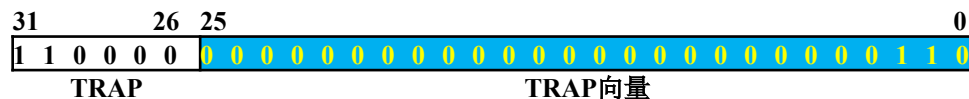


- **访问内存**

- MAR ← x0000 0018
- 读取存储器
- MDR ← x0000 2500

x0000 0000-x0000 0003	xFFFE 0100
⋮	⋮
x0000 0018-x0000 001B	x0000 2500
x0000 001C-x0000 001F	x0000 2900
x0000 0020-x0000 0023	x0000 2D00
x0000 0024-x0000 0027	x0000 3100
x0000 0028-x0000 002B	x0000 3500
⋮	⋮

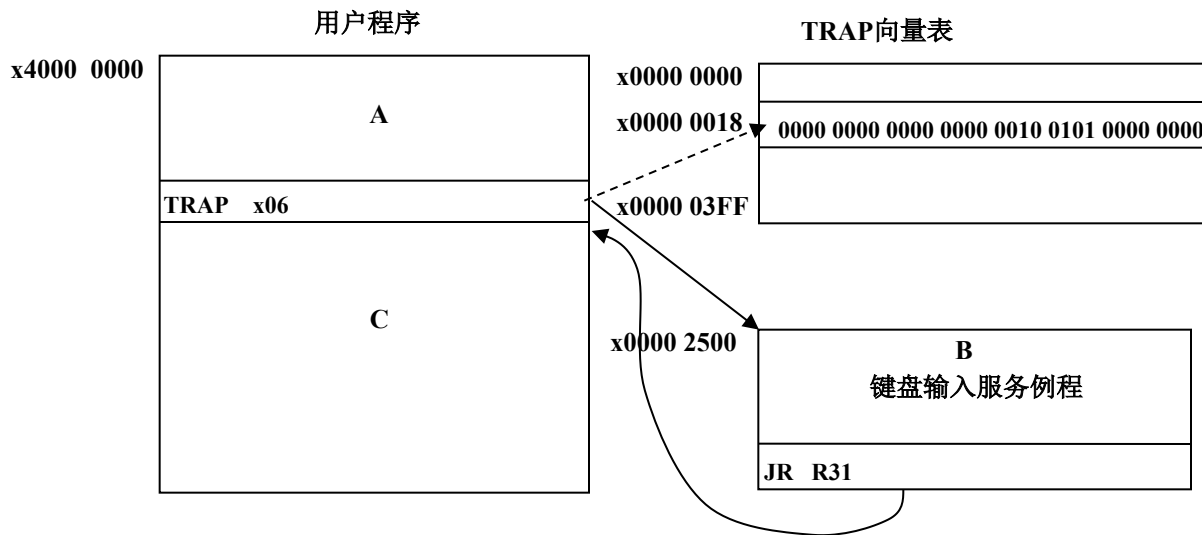
# TRAP指令（5阶段）



- 写回
  - **R31** ← PC
    - 返回用户程序的“链接”
  - **PC** ← MDR
    - x0000 2500，字符输入服务例程的起始地址

# JR R31

- **TRAP服务例程执行结束**
  - 在TRAP服务例程的最后执行一条JR R31指令，控制就可以返回到用户程序的正确位置



# 小写→大写

```

                .TEXT
                .GLOBAL MAIN
MAIN :          TRAP    x09                ; 请求键盘输入
                SEQI    R1, R4, x0A        ; 测试是否是终止字符
                BNEZ    R1, EXIT
                SUBI    R4, R4, x20        ; 改变为大写字母
                TRAP    x07                ; 输出到显示器
                J        MAIN              ; 再做一次
EXIT :          TRAP    x00                ; 停止
```

- 模拟器：Step Over/Into
  - 跟踪trap指令，查看PC、R31等值的变化

# I/O服务例程

- 在伪操作.text中说明服务例程的起始地址
  - TRAP向量表
- 在.data中说明的地址是数据区的地址
- 使用JR R31结束这个输入服务例程

# 键盘输入服务例程 (TRAP x06)

01		.data	x0000 2400	
02	.....			
03	.....			
04	.....			
05	KBSR:	.word	xFFFF0000	; KBSR的起始地址
06	KBDR:	.word	xFFFF0004	; KBDR的起始地址
.....				
07		.text	x0000 2500	
08	.....			
09	.....			
0A	.....			
0B		lw	r1, KBSR(r0)	
0C	INPOLL:	lw	r2, 0(r1)	; 测试是否有字符被输入
0D		andi	r3, r2, #1	
0E		beqz	r3, INPOLL	; 如果KBSR[0]==0, 轮询
0F		lw	r1, KBDR(r0)	
10		lw	r4, 0(r1)	
11	.....			
12	.....			
13	.....			
14		jr	r31	; 返回用户程序

- 11行
  - R1=xFFFF0004, R2=1, R3=1



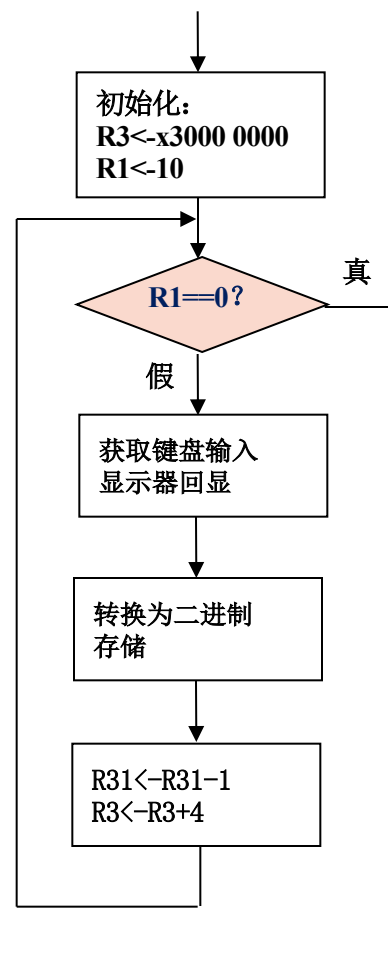
# 字符输出服务例程（TRAP x07）

01			.data	x00002800	
02	.....				
03	.....				
04	DSR:	.word	xFFFF0008	; DSR的起始地址	
05	DDR:	.word	xFFFF000C	; DDR的起始地址	
06	;				
07			.text	x00002900	
08	.....				
09	.....				
0A		lw	r1, DSR(r0)		
0B	OUTPOLL:	lw	r2, 0(r1)		; 测试输出寄存器是否就绪
0C		andi	r2, r2, #1		
0D		beqz	r2, OUTPOLL		
0E		lw	r1, DDR(r0)		
0F		sw	0(r1), r4		
10	.....				
11					
12		jr	r31		; 从TRAP返回

- 10行
  - R1=xFFFF000C, R2=1

# ASCII → 二进制

- 输入10个数字字符  
(一位数)
- 转换为整数，存储
- 寄存器
  - R1，计数器
  - R3，指针



# ASCII → 二进制, 是否有问题?

```
01      .data      x30000000
02 Binary:      .space      40
03 ;
04      .text      x40000000
05 .....
06      addi      r3, r0, Binary
07      addi      r1, r0, #10
08 AGAIN:      beqz      r1, EXIT
09      trap      x06
0A      trap      x07
0B      subi      r4, r4, x30
0C      sw        0(r3), r4
0D      addi      r3, r3, #4
0E      subi      r1, r1, #1
0F      j         AGAIN
10 EXIT: .....
```

R1=xFFFF0004,  
R2=1, R3=1

; R4得到键盘输入的字符  
; 回显R4中的字符  
; ASCII码-> 二进制

; 指针加4

R1=xFFFF000C,  
R2=1

# 寄存器的保存与恢复

- 如果一个寄存器内的值在该寄存器被存储了其他值之后再次用到，必须在其他事情发生之前将其**保存**，在再次使用它之前将其**恢复**

# 被调用者保存

- **callee-save**
- 由被用户程序调用的服务例程完成寄存器的保存与恢复
  - 被调用程序知道需要使用哪些寄存器，而调用者不知道哪些寄存器的值将被破坏

# 键盘输入服务例程 (TRAP x06)

01		.data	x0000 2400	
02	SaveR1:	.space	#4	; 保存寄存器的存储单元
03	SaveR2:	.space	#4	
04	SaveR3:	.space	#4	
05	KBSR:	.word	xFFFF0000	; KBSR的起始地址
06	KBDR:	.word	xFFFF0004	; KBDR的起始地址
.....				
07		.text	x0000 2500	
.....				
08		sw	SaveR1(r0), r1	; 保存此例程需要的寄存器
09		sw	SaveR2(r0), r2	
0A		sw	SaveR3(r0), r3	
0B		lw	r1, KBSR(r0)	
0C	INPOLL:	lw	r2, 0(r1)	; 测试是否有字符被输入
0D		andi	r3, r2, #1	
0E		beqz	r3, INPOLL	; 如果KBSR[0]==0, 轮询
0F		lw	r1, KBDR(r0)	
10		lw	r4, 0(r1)	
11		lw	r1, SaveR1(r0)	; 将寄存器恢复为原先的值
12		lw	r2, SaveR2(r0)	
13		lw	r3, SaveR3(r0)	
14		jr	r31	; 返回用户程序

# 字符输出服务例程 (TRAP x07)

01		.data	x00002800	
02	SaveR1:	.space	4	; 保存寄存器的存储单元
03	SaveR2:	.space	4	
04	DSR:	.word	xFFFF0008	; DSR的起始地址
05	DDR:	.word	xFFFF000C	; DDR的起始地址
06	;			
07		.text	x00002900	
08		sw	SaveR1(r0), r1	; 保存此例程需要的寄存器
09		sw	SaveR2(r0), r2	; 在返回之前被恢复
0A		lw	r1, DSR(r0)	
0B	OUTPOLL:	lw	r2, 0(r1)	; 测试输出寄存器是否就绪
0C		andi	r2, r2, #1	
0D		beqz	r2, OUTPOLL	
0E		lw	r1, DDR(r0)	
0F		sw	0(r1), r4	
10		lw	r1, SaveR1(r0)	; 将寄存器恢复为原先的值
11		lw	r2, SaveR2(r0)	
12		jr	r31	; 从TRAP返回

# ASCII → 二进制, R1改为R31, 无限循环!

```
01      .data      x30000000
02 Binary:      .space      40
03 ;
04      .text      x40000000
05 .....
06      addi      r3, r0, Binary
07      addi      r31, r0, #10
08 AGAIN:      beqz      r31, EXIT
09      trap      x06
0A      trap      x07
0B      subi      r4, r4, x30
0C      sw        0(r3), r4
0D      addi      r3, r3, #4
0E      subi      r31, r31, #1
0F      j         AGAIN
10 EXIT: .....
```

R31, 0A行的地址

; R4得到键盘输入的字符  
; 回显R4中的字符  
; ASCII码-> 二进制

; 指针加4

R31, 0B行的地址



# 调用者保存

- caller-save
- 由调用程序完成寄存器的保存与恢复

01		.data	x30000000	
.....				
	SaveR31:	.space	#4	; Caller-Save
.....				
04		.text	x40000000	
.....				
07		addi	r31, r0, #10	
08	AGAIN:	beqz	r31, EXIT	
		sw	SaveR31(r0), r31	; Caller-Save
09		trap	x06	
0A		trap	x07	
		lw	r31, SaveR31(r0)	; Caller-Save
.....				
0E		subi	r31, r31, #1	
0F		j	AGAIN	
10	EXIT:	.....		

# 保存/恢复

- 通过：
  - 在TRAP执行之前由调用程序处理
    - 由调用程序处理这个问题，被称为caller-save（调用者保存）
  - 或者在TRAP指令执行之后由被调用的程序（例如，服务例程）处理
    - 由被调用的程序处理这个问题，被称为callee-save（被调用者保存）

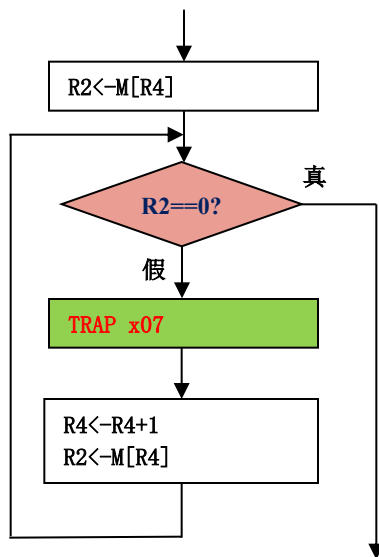
# 保存/恢复

- 原则

- 如果哪个程序知道哪些寄存器将被接下来的操作所破坏，处理保存/恢复问题的就应该是哪一个程序

# 输出字符串服务例程（TRAP x08）

- 在显示器上显示一串字符，该字符串以x00结束
  - 在R4中提供该字符串的起始地址，调用TRAP x08
- 标志控制的循环



# 输出字符串服务例程

```
01      .data      x00002C00
02  SaveR2:      .space      4
03  SaveR4:      .space      4
04  SaveR31:     .space      4
05  CallerSR4:   .space      4
06  ;
07      .text      x00002D00
08      sw        SaveR2(r0), r2      ; Callee-Save
09      sw        SaveR4(r0), r4
0A      sw        SaveR31(r0), r31    ; Caller-Save
0B  ;
0C  ; 对字符串中的每一个字符进行循环
0D  ;
0E  LOOP:      lb        r2, 0(r4)    ; 取得字符
0F              beqz     r2, Return    ; 如果是0, 字符串结束
10              sw        CallerSR4(r0), r4    ; Caller-Save
11              addi     r4, r2, #0
12              trap     x07
13              lw        r4, CallerSR4(r0)
14              addi     r4, r4, #1      ; 指针加1
15              j        LOOP          ; 获取下一个字符
16  ;
17  ; 从服务调用请求返回
18  Return:     lw        r1, SaveR1(r0)
19              lw        r4, SaveR4(r0)
1A              lw        r31, SaveR31(r0)
1B              jr       r31          ; 从TRAP返回
```

# 被调用者保存

- **callee-save**
- **R2**, 获取的下一个字符
- **R4**, 下一个字符地址

# 调用者保存

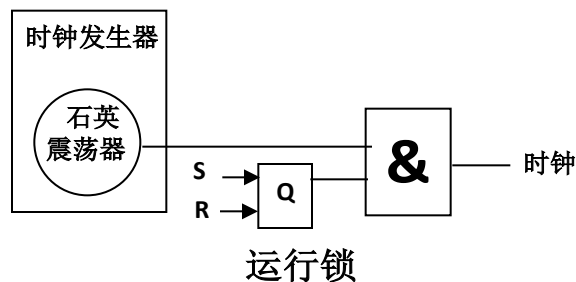
- **caller-save**
- **R31, 返回链接**
  - 在此服务例程中, 又调用了trap指令
- **R4, 下一个字符地址, 要输出的字符**
  - **Trap x07, 输出的R4中的字符**

# 寄存器的保存/恢复

- 一个寄存器的值，在该寄存器被存储了其他值之后再次用到
  - 在使用之前，将其保存
  - 再次使用之前，将其恢复
- 采用调用者/被调用者保存机制的原则
  - 哪个程序知道哪些寄存器将被接下来的操作所破坏，处理保存/恢复问题的就是哪一个程序



# 停机服务例程（TRAP x00）



- 运行锁
  - MCR[0], （机器控制寄存器）
  - 内存映射地址, xFFFF 00F8

# 停机服务例程

01		.data	xFFFE0000	
02	SaveR1:	.space	4	; 保存寄存器的存储单元
03	SaveR2:	.space	4	
04	MCR:	.word	xFFFF00F8	; MCR的内存映射地址
05				;
06		.text	xFFFE0100	
07		sw	SaveR1(r0), r1	; 保存此例程需要的寄存器
08		sw	SaveR2(r0), r2	
09				;
0A				; 清空xFFFF 00F8的0位, 停机
0B				;
0C		lw	r1, MCR(r0)	
0D		lw	r2, 0(r1)	; 加载MCR的值到R2中
0E		andi	r2, r2, #-2	; 清空MCR的[0]位
0F		sw	0(r1), r2	; 将R2的值存储到MCR中
10				;
11				; 从服务例程返回
12				;
13		lw	r1, SaveR1(r0)	; 将寄存器恢复为原先的值
14		lw	r2, SaveR2(r0)	
15		jr	r31	; 从TRAP返回

# IN服务例程（TRAP x09）

- 将键盘输入服务例程（TRAP x06）稍做修改
- 1、输出提示符
- 2、键盘输入→R4
- 3、回显
- 4、输出新行符

# 中断驱动的I/O

- 本质是I/O设备能够：
  - (1) 强制程序停止
  - (2) 让处理器执行I/O设备的请求
  - (3) 让停止的程序继续执行，好像什么都没发生过
    - 程序A执行指令n；
    - 程序A执行指令n+1；
    - 程序A执行指令n+2；
    - (1) 检测到中断信号；
    - 程序A进入**休眠**状态；
    - (2) 执行I/O设备的请求；
    - (3) 程序A被**激活**；
    - 程序A执行指令n+3；
    - 程序A执行指令n+4；

# 机制

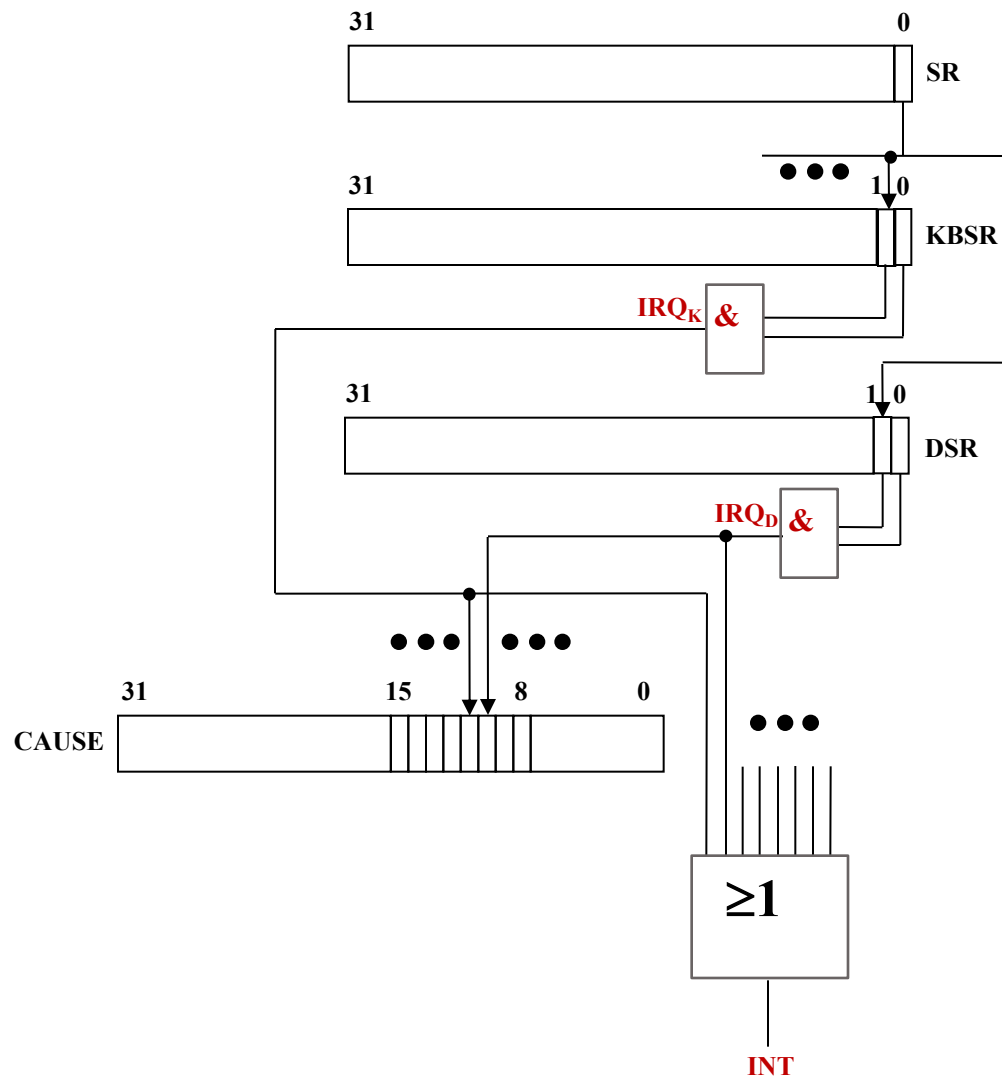
- 1、当I/O设备有输入要处理，或准备接受输出时，允许I/O设备中断处理器的机制
- I/O控制器：
  - 生成中断信号INT
- 2、管理I/O数据传送的机制
- 处理器：
  - 停止当前的执行过程
  - 处理由该信号发出的请求

# 中断信号的产生

- 必须满足以下两个元素：
  - I/O设备必须需要服务
    - KBSR或DSR的**就绪位**，当相应的就绪位被设为1时，I/O设备就需要服务
  - 设备必须有权去请求服务
    - **中断允许位（IE）**
    - 可以被设置为1或清为0，取决于是否给I/O设备权利去请求服务
    - 在大多数I/O设备里，中断允许位是设备状态寄存器的一部分

# INT信号

- KBSR和DSR
  - IE位是[1]位
- 中断请求信号（Interrupt Request, **IRQ**）
  - IE位和、就绪位的**逻辑与**
- 各中断设备发出的IRQ信号经过**或门**，产生INT信号





# 原因寄存器

- 如果某个设备发出IRQ信号，就会将一个被称为原因寄存器（**CAUSE**）的相应位设为1
  - **记录**哪些设备发出中断信号
  - DLX的一个**特殊寄存器**，只有在**特权模式**下（操作系统），才能访问
  - **CAUSE[15:8]**为**中断未决位**

# 状态寄存器

- 所有设备的IE位的信号可以被状态寄存器**SR[0]**同时改写
  - DLX的一个**特殊寄存器**，只能在特权模式下访问
  - **SR[0]**表示**中断允许位**，决定了谁能中断处理器
    - 如果**SR[0]**为0，那么所有I/O设备都不能中断处理器，在这种情况下，只能采用轮询方式访问I/O设备
    - 如果**SR[0]**位被设置为1，那么允许所有I/O设备中断

# 测试INT信号

- 发出INT信号后，**处理器**如何发现这个信号？
  - 指令执行按顺序为取指令、译码、执行、访问内存和写回5个阶段
  - 为测试中断信号而**增加逻辑**
    - 将总是从写回返回到取指令的最后一步取代为：写回，并检测INT信号
      - 如果INT信号为0，那么它与往常一样，控制单元将返回取指令阶段，开始下一条指令的处理
      - 如果INT信号为1，
        - **保存及改变程序状态**
        - 那么控制单元将PC加载为**x8000 1000**，执行操作系统的**中断服务例程**，处理由该信号发出的中断请求

# 保存及改变程序状态

- 中断服务例程类似于自陷服务例程
  - 存储在存储器的一些预先分配的单元中的程序片段，为中断请求服务
- 在进入中断服务例程之前（PC加载为x8000 1000之前）
  - 保存足够的正在运行的程序的状态信息，以便当I/O设备请求被满足之后，能够返回被中断的程序
  - 改变程序状态，以便访问恰当的资源，以及避免各种I/O设备互相干扰

# 程序状态

- 程序影响的所有资源所包含的内容的瞬态图，包括
  - 作为程序一部分的**存储单元**的内容
  - 所有**通用寄存器**的内容
  - **寄存器**：PC和SR

# PC和EPC

- 因为PC包含了下一条要执行的指令的地址，必须被保存起来，以便当被中断的程序重新执行时，可以正确的返回到下一条指令地址
- DLX有一个特殊寄存器EPC，用于保存中断发生时PC中的值

# SR[0]

- 程序是否可以被I/O设备中断，应该被保存
  - 例如，用户程序允许被中断，而进入中断服务例程，为避免受到来自其他设备的中断信号的干扰，则应屏蔽所有中断，即SR[0]从1改变为0
  - 因此，SR[0]需要保存起来，以便返回到用户程序时，仍可被I/O设备中断

# SR[1]

- 表示正在运行的程序是处于**特权**（管理员或内核）模式还是**非特权**（用户）模式
  - 特权模式为0，用户模式为1
  - 在特权模式下，可以访问对用户程序不可见的重要资源，如CAUSE寄存器



# SR[1]

- 程序的特权级别包含了被中断的程序能够访问哪些资源，禁止访问哪些资源，必须被保存
  - 如果从用户程序进入中断服务例程，SR[1]就**从1改为0**，因为中断服务例程需要访问SR、CAUSE、EPC等寄存器，因此，SR[1]也**需要保存**起来
- 当中断发生时，DLX使用SR[2]保存SR[0]的值，使用SR[3]保存SR[1]的值，即利用SR实现了一个**硬件栈**

# 中断服务例程

- 首先，对于一个I/O设备真正能中断处理器，要求来自设备的请求必须比处理器当前的工作更紧急
  - 执行的紧急程度被称为**优先级**
- 服务该中断
  - 进入设备处理例程
- 最后，从中断返回

# 中断优先级

- 为了让I/O设备成功的停止处理器，开始中断请求的处理，请求的优先级必须比它希望中断的程序更高
- DLX有6个**硬件优先级**，PL0, ... ..., PL5
  - 数字越高，优先级越高
  - 速度越高的I/O设备，优先级也越高
    - 例如，键盘优先级别为1，显示器级别为0
  - 在最低的优先级下，允许所有中断，在最高的优先级下，则屏蔽所有中断

# SR[15:8]

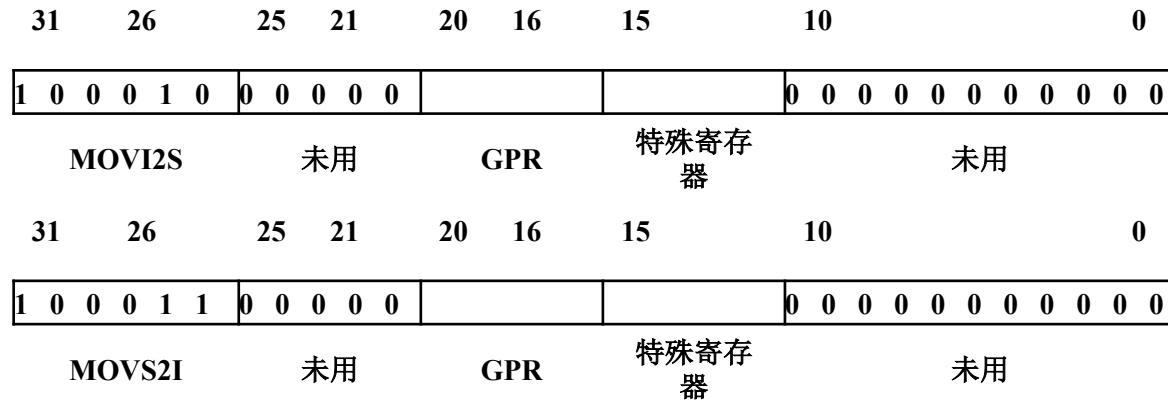
- DLX的中断优先级，采用状态寄存器SR[15:8]表示
- SR[15:8]给出了**中断阻塞方案**，以决定系统响应哪些中断
  - 与CAUSE[15:8]一一对应
  - 从左至右，优先级别依次降低
  - 要允许某一级别的中断，屏蔽位必须为1
  - 原因寄存器中的未决中断要等到相应的屏蔽位为1时，才能引起处理器的处理

# 中断服务例程

- 第一项任务
  - 对原因寄存器的中断未决位CAUSE[15:8]和状态寄存器的中断掩码位SR[15:8]做“**逻辑与**”运算，看发生了哪些允许的中断
  - 如果有多于1个的允许中断发生，则选择**优先级高**的中断（左边的优先级更高）

# MOVI2S和MOVS2I

- DLX的数据传送指令
  - 整数寄存器和特殊寄存器之间进行数据传送



寄存器名称	编号		用途
	D	H	
SR	12	x0C	中断屏蔽，中断允许位
CAUSE	13	x0D	未决中断位
EPC	14	x0E	包含了中断发生时的PC中的值

# 代码

；检查发生了哪些允许的中断

movs2i        r1, x0D

；将**原因寄存器**的值送至R1中

movs2i        r2, x0C

；将**状态寄存器**的值送至R2中

movs2i        r6, x0E

；将**EPC**的值送至R6中

andi    r3, r1, xFF00; CAUSE[15:8], 假设CAUSE[31:16]均为0

andi    r4, r2, xFF00; SR[15:8], 假设SR[31:16]均为0

and     r3, r3, r4                    ; CAUSE[15:8] & SR[15:8]

beqz    r3, DONE        ; 没有允许的中断

... ..

；按照优先级规则，依次调用设备处理例程

**TEST5:**        **slli**     **r3, r3, #16**

**slti**     **r5, r3, #0**

**addi**   **r7, r0, TEST4**

**bnez**   **r5, DEV5**

；优先级为PL5的设备

**TEST4:**        **slli**     **r3, r3, #1**

**slti**     **r5, r3, #0**

**addi**   **r7, r0, TEST3**

**bnez**   **r5, DEV4**

；优先级为PL4的设备

.....

**TEST1:**        **slli**     **r3, r3, #1**

**slti**     **r5, r3, #0**

**addi**   **r7, r0, TEST0**

**bnez**   **r5, DEV1**

；键盘， PL1

**TEST0:**        **slli**     **r3, r3, #1**

**slti**     **r5, r3, #0**

**bnez**   **r5, DEV0**

；显示器， PL0



# 服务该中断

； 键盘

DEV1:lw        r1, KBDR(r0)

         lw        r4, 0(r1)                                ； 将输入的字符加载到R4

         jr        r7

- 此时，键盘控制器将键盘状态寄存器的就绪位清空，表示已处理完该字符

# 从中断返回

- 首先，**清空CAUSE**寄存器，表明处理完所有的中断
- 然后，使用**RFE指令**
  - 将**PC恢复**为EPC中的值，即假设程序没有被中断的下一条执行的指令地址
  - 将**SR[0]恢复**为SR[2]的值，将SR[1]恢复为SR[3]的值，即允许中断和返回用户模式

# 代码

## ; 从中断返回

**DONE:**      **movi2s**      **r0, x0D**      ; 将原因寄存器清空

**rfe**

31  
26

25

# O

[illegible]

RFE

未用

# 中断嵌套

- 在中断服务例程中执行键盘处理例程时，如果允许被比键盘优先级高的设备所中断
  - 以键盘处理例程为例，在读取KBDR的值之前，要先执行：
    - 1) 保存SR和EPC的值
    - 2) 将SR[15:8]的值改为xF0，即SR[15:12]均为1，SR[11:8]均为0，也就是屏蔽比该设备优先级低（或相等）的其他设备的中断，允许优先级高的设备的中断
    - 3) 将SR[0]改为1，即允许中断。这样，就允许被其他优先级高的设备所中断。

- 因为改变了SR[15:8]和SR[0]的值，在结束键盘处理例程之前，应先将SR[0]改为0（因为接下来恢复SR和EPC的过程不能被中断），再恢复SR和EPC的值
- 需要使用栈结构来存储程序状态

# C中的I/O

- **C程序中的输入和输出，是通过库函数执行的，这一点与汇编程序通过调用TRAP指令，执行服务例程十分类似**

# putchar/getchar

- 对**单个字符**执行输出和输入
  - getchar: 读取一个ASCII码
  - putchar: 写一个ASCII码
  - 与TRAP服务例程类似
- 头文件stdio.h

# putchar

- 与显示器输出服务例程类似
- 不执行类型转换——传给它的值假定为ASCII码



# 示例

```
char c = 'h';  
putchar (c);  
putchar ('h');  
putchar (104);
```

# getchar

- 与键盘输入服务例程类似
- 返回一个从键盘上键入的字符的ASCII值

```
char c;  
c = getchar ();
```

# 示例

```
#include <stdio.h>

int main()
{
    char inChar1;
    char inChar2;

    printf ("Input character 1:\n");
    inChar1 = getchar ( );

    printf ("Input character 2:\n");
    inChar2 = getchar ( );

    printf ("Character 1 is %c\n", inChar1);
    printf ("Character 2 is %c\n", inChar2);
}
```

Input character 1:

A回车

Input character 2:

Character 1 is A

Character 2 is

# I/O流

- 现代程序设计语言为考虑I/O创造了一个有用的**抽象**：输入和输出发生在流上
  - 基于**字符**的I/O
- 输入流
  - 键盘
    - 一个字符被键入，添到流的结尾处
    - 程序读取输入，从流的开头处读
- 输出流
  - 打印机
    - 程序打印的字符，添到输出流的结尾处
    - 打印机，从输出流的开头处打印

# 生产者/消费者

- 生产者，添加数据到流中
- 消费者，从流中读取数据
- 两者以不同速率运转

# 生产者和消费者

- 流的抽象
  - 允许把生产者和消费者分开
  - 二者以其各自的速率操作，不用等待另一个是否就绪

# 使用流的原因

- **I/O设备和CPU，两者通常以不同的速率运转**
  - **流，又称缓冲区，用于缓存数据**
    - 如果一个程序想要执行某些输出，它把字符添加到输出流的结尾处即可，而不必等待输出设备结束前一个字符的输出
- **使低速的输入输出设备和高速的CPU能够协调工作，避免低速的输入输出设备占用CPU，解放出CPU，使其能够高效率工作**

# 输出流的例子

- 一个程序使用打印机打印文档
  - 打印机的打印速度较慢
- 程序：把文档添加到打印机输出流中
  - 生产者
  - 不必等待打印机结束前一个字符的打印
- 打印机：从输出流中打印字符
  - 消费者
  - 打印时，计算机/CPU可以处理其他任务



# 输入流的例子

- 输入设备：把字符添加到输入流中
  - 生产者
  - 不必等待程序准备读取数据
- 程序：从输入流中读取字符
  - 消费者

# stdin/stdout

- 标准输入流
  - `stdin`, 缺省映射到键盘
  - `getchar`, 返回`stdin`中的下一个输入ASCII码
- 标准输出流
  - `stdout`, 缺省映射到显示器
  - `putchar`, 把传递给它的ASCII码添加到`stdout`中
- C++, 相似的基于流的抽象

# 键盘缓冲区

- 原因之一
  - 回车键，允许**用户确认**输入
    - 假如错按又想改正，使用退格键
    - 回车键确认输入
- 1、每个键盘上的输入 → 键盘缓冲区
- 2、用户按下回车键，键盘缓冲区 → 输入流
  - **回车键**作为**换新行字符** → 输入流
  - 键盘缓冲区被清空

# 示例

```
#include <stdio.h>

int main()
{
    char inChar1;
    char inChar2;

    printf ("Input character 1:\n");
    inChar1 = getchar ( );

    printf ("Input character 2:\n");
    inChar2 = getchar ( );

    printf ("Character 1 is %c\n", inChar1);
    printf ("Character 2 is %c\n", inChar2);
}
```

Input character 1:

A回车

Input character 2:

Character 1 is A

Character 2 is

stdin: 65,10

inChar1: 'A'

inChar2: '\n'

# I/O流的实现

- 通过包围在I/O服务例程之外的额外的软件层实现

# 流的缓冲

- 键盘和显示器：行缓冲设备
- 键盘
  - 按键→内核的键盘缓冲区（FIFO，16个字节）
  - 遇回车
    - 键盘缓冲区→用户区的数据缓冲区（“输入流”，堆中，4096字节，与“页”大小一致）
  - 程序从数据缓冲区读数据

# 其他

- **stdin, stdout**
  - **FILE \*类型（附2）**
- **程序写文件到磁盘（输出）**
  - **Data→用户区的流缓冲区（I/O库函数实现的，使用 malloc()在堆中分配空间，环形队列数据结构）**
  - **遇新行'\n'**
    - 流缓冲区→内核缓冲区（因为磁盘是**块设备**，需要再整理一次，OS实现的）
    - 如缓冲区满→磁盘
- **I/O缓冲区（用户区的）**
  - **被生产者/消费者共享问题，OS使用信号量解决**
    - 如，程序（消费者）读键盘（生产者）
    - 当缓冲区无数据，程序如何读？——中断机制

# printf/scanf

- 执行非ASCII码的I/O任务
  - 复杂的格式化I/O
  - 整数和浮点型数值的输入和输出



# printf

- 将格式化文本写进输出流中
- 考虑了所有必需的**类型转换**

# printf函数

- 把**格式用字符串**写到输出流中去
  - 转换说明, “%” 开头
    - %d, 将一个二进制补码整数转化为ASCII码字符序列, 写到输出流中
  - 特殊字符, “\” 开头
    - \n, 新行符
    - \t, 制表符
  - 如果字符不是 “%” 或 “\”, 字符→输出流

# printf

- 问题：
- 如何打印出一个 “%” 字符本身？
  - “%%”
- 如何打印出一个反斜杠字符？
  - “\\”

# printf

```
printf ("25 plus 76 in decimal is %d. \n", 25 + 76);  
printf ("25 plus 76 in hexadecimal is %x. \n", 25 + 76);  
printf ("25 plus 76 in octal is %o. \n", 25 + 76);  
printf ("25 plus 76 as a character is %c. \n", 25 + 76);
```

```
25 plus 76 in decimal is 101.  
25 plus 76 in hexadecimal is 65.  
25 plus 76 in octal is 145.  
25 plus 76 as a character is e.
```

# printf

- **%f**
  - 将浮点数转化为如 “3.140000”形式的字符序列
- **注意**
  - 在转换说明和数值类型之间没有关系
  - 程序员可以自由选择如何解释这些数值

# 显示器缓冲区

- 向输出流中写新行符 “\n”
  - 输出流 → 显示器
- 此前，输出流中的内容，可能还不会出现在显示器上

# scanf

- 从输入流中读入一个格式化的ASCII码文本
- 格式用字符串
  - 包含：文本和转换说明
  - %d
    - 将输入流中的一个用十进制计数法表示的整数的ASCII字符序列，转换为二进制整数

# 示例

```
int month, day, year;  
double gpa;  
  
printf ("Enter: birthday grade_point_average\n");  
scanf ("%d/%d/%d %lf", &month, &day, &year, &gpa);
```

- **%d**
  - 从标准输入流的一个非空白字符开始，找到多位数字（至少一位）
    - 空白字符：空格、制表符、新行、回车、垂直制表符和换页
  - 抛弃掉空白字符，读入十进制数，以“非数字”结束
  - 转化为二进制整数
- **/**
  - 期望在输入流中找到
  - 被抛弃
- **%lf**
  - 输入流，一串十进制数字
  - 可以包括第一个E或e，或“.”，或“+”，或“-”，以非数字结束
  - 转换成双精度浮点数



# 示例

```
int month, day, year;
```

```
double gpa;
```

```
printf ("Enter: birthday grade_point_average\n");
```

```
scanf ("%d/%d/%d %lf", &month, &day, &year, &gpa);
```

- 正确的输入


08/08/90 4.15

08/08/90

4.15

# 示例

```
int month, day, year;  
double gpa;  
  
printf ("Enter: birthday grade_point_average\n");  
scanf ("%d/%d/%d %lf", &month, &day, &year, &gpa);
```

- 如果输入：08 08 90 4.15
- $\text{month} \leftarrow 8$
- 剩下的变量，不被赋值
- scanf函数的返回值
  - 在输入流中成功扫描的格式说明的个数
- 注意：没有被使用的输入不会被丢弃，仍然留在输入流中

# 示例

```
int month, day, year;  
double gpa;  
char a, b;  
  
printf ("Enter: birthday grade_point_average\n");  
scanf ("%d/%d/%d %lf", &month, &day, &year, &gpa);  
  
a = getchar ();  
b = getchar ();
```

- 如果输入：

08 08 90 4.15
- 问题：a和b将包含什么？
  - 从输入流中前一次调用离开的地方开始读取
  - 答案：空格和0

# scanf格式说明

- “%d” “%lf”
  - 从标准输入流的一个非空白字符（抛弃掉之前的空白,无论这个非空白是否为数字）开始，找到多位数字（至少一位），以“非数字”（不抛弃，仍在输入流中）结束

# printf

- **%u**——无符号数输出（根据其二进制表示）；
- **%d**——按实际长度输出整数；
  - **%md**——指定宽度输出，如果实际位数<m，左端补空格；如果实际位数>m，按实际长度输出；
  - **%-md**——如果实际位数<m，右端补空格；
  - **%mo**， **%mx**， **%mu**与之类似；
  - **%\*d**——printf(“%\*d\n”,m,x); //m为宽度，x为输出的整数
- **%f**——输出6位小数；
  - **%m.nf**——m列，n位小数，左端补空格；
  - **%-m.nf**——右端补空格；
  - **%.nf**——输出double型数时，n>6
- **%lf** 或 **%Lf**——输出6位小数；
- **%e**或**%E**——按指数形式输出
  - 小数点前1位，小数点后6位小数，e占1位，符号占1位，指数占3位；
  - 3.333333e+002 或 3.333333E+002

# scanf

- **%md** ——自动截取m位整数；
- **%f** ——读入float型，取7位数；
  - **%m.nf** 无法读入数据；
- **%lf 或 %Lf** ——读入double型数；
- **%\*md** ——跳过m位整数；
  - **%\*mf** ——跳过m列小数；
  - **%\*mc** ——跳过m个字符；
- **%x, %o** ——读入十六进制、八进制数
- **“%d%d%d ”** ——数据间以空格、Tab、回车键间隔
- **“%d,%d”** ——数据间**原样输入** “逗号”
- **“%2d %\*3d %2d”,&a,&b**
  - ——输入12 345 67，则a=12,b=67

# 习题

- 13.3
- 13.4
- 13.5(订正：6改为24)
- 13.7
- 13.10
- 13.13
- 13.14