

第十四章 子例程

子例程

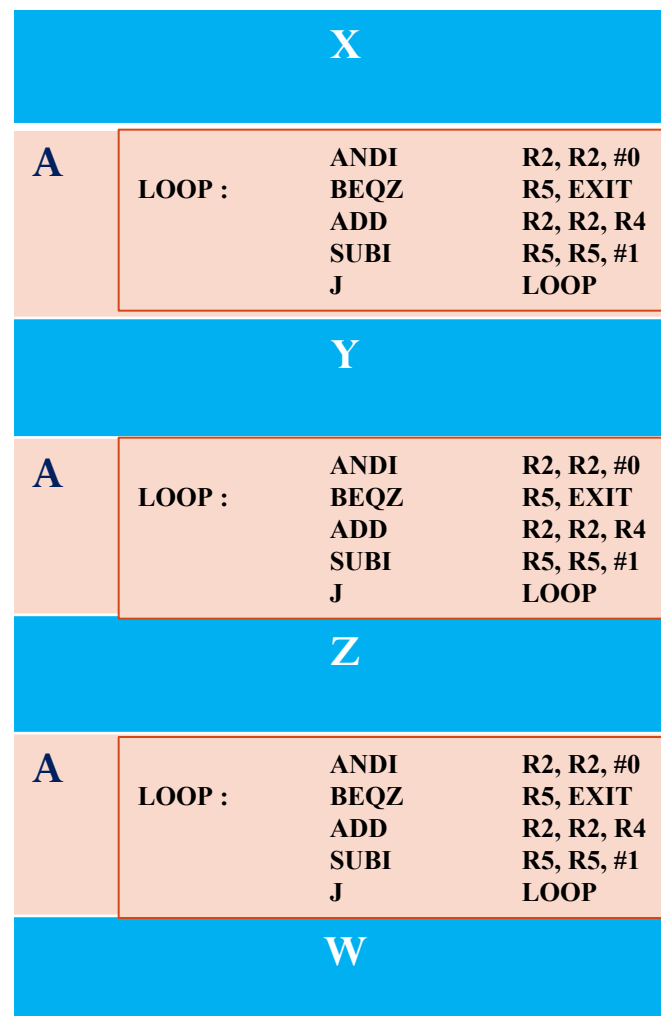
- 在一个程序中，**多次执行某个程序片段**
 - 在程序内，不必每次说明其源程序段的全部细节
 - 通过**多次调用**该程序片段实现
- 子例程/函数（C语言）
- 可以由不同的程序员**分别实现**

库

- 由某个独立的软件供应商提供
 - 程序片段的集合
 - 数学库，执行平方根、正弦、余切等

子例程机制

- 调用/返回机制
- 程序片段A
 - $R2 \leftarrow R4 * R5$



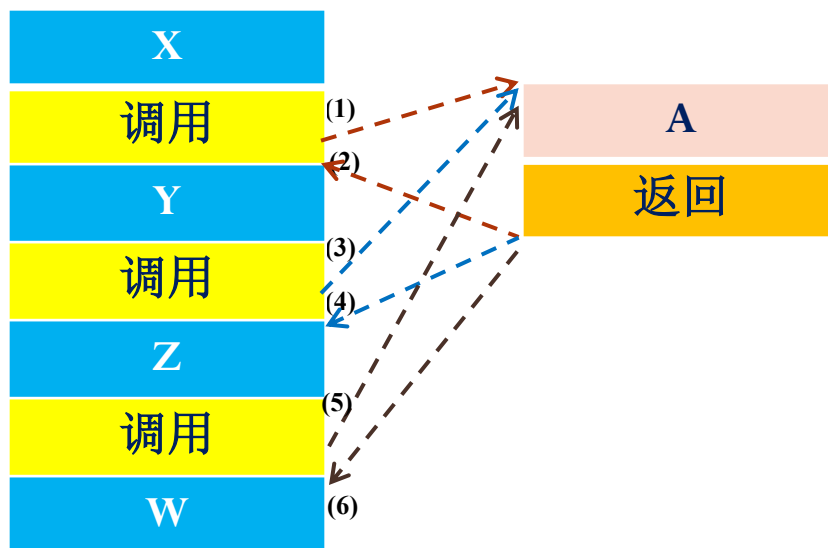
调用/返回机制

- **调用机制**

- 计算子例程的起始地址，加载到PC，并保存返回地址

- **返回机制**

- 使用返回地址加载PC



与TRAP指令相似

- 与TRAP指令相似
 - 将PC加载为程序片段的起始地址，同时R31被加载返回地址
 - 程序片段的最后一条指令
 - JR R31

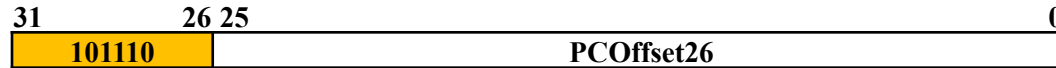
与TRAP指令的区别

- TRAP指令
 - 服务例程包括操作系统资源
 - 由系统程序员编写
- 子例程
 - 由相同程序员
 - 或某个同事编写
 - 或某个库

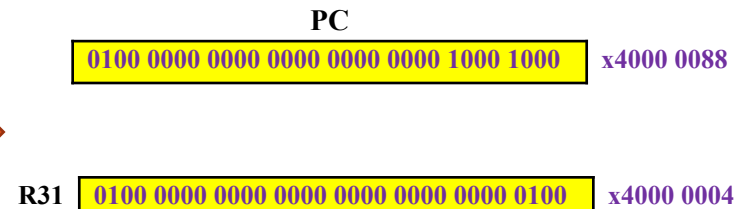
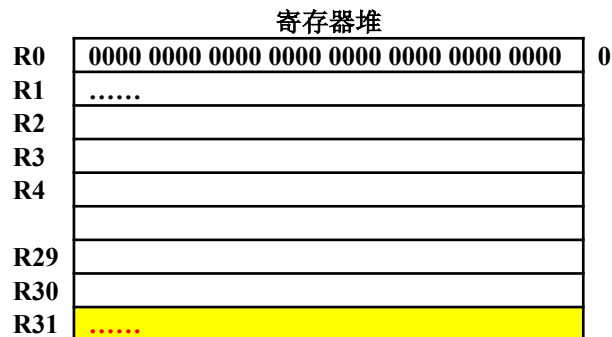
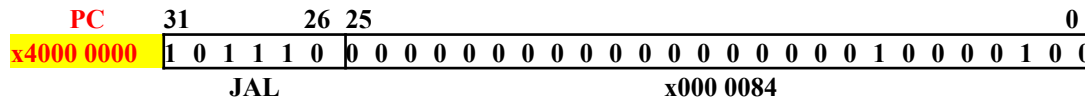
JAL和JALR指令

- 调用子例程指令
 - 2种方法，计算子例程的起始地址
 - JAL: Jump and Link
 - JALR: Jump and Link Register
- JAL和JALR指令
 - 在R31中保存返回地址
 - PC+4
 - 计算子例程的起始地址并加载到PC

JAL



- $PC \leftarrow PC + 4 + \text{SEXT}(\text{PCOffset26})$
 - 与J指令相同
- $R31 \leftarrow \text{返回地址}$



JAL指令汇编格式

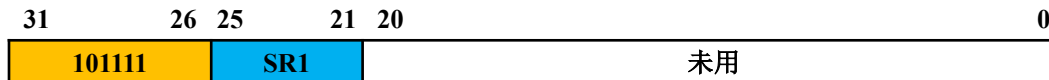
JAL LABEL

- **LABEL**，标识JAL指令的目标
- **例如，JAL Multiply**
 - **下一条被执行的指令**
 - 被Multiply标识的指令
 - 并且，在R31中保存返回地址

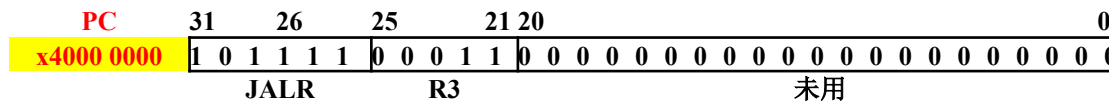
Multiply子例程

```
Multiply:  ANDI  R2, R2, #0
LOOP :      BEQZ  R5, EXIT
            ADD   R2, R2, R4
            SUBI  R5, R5, #1
            J     LOOP
EXIT:       JR    R31
```

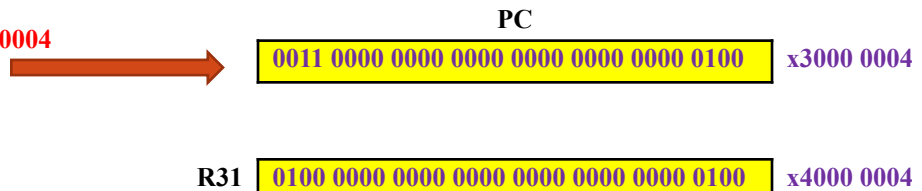
JALR



- $PC \leftarrow (SR1)$
 - 与JR指令相同
- $R31 \leftarrow$ 返回地址



寄存器堆	
R0	0000 0000 0000 0000 0000 0000 0000 0000 0
R1
R2	
R3	0011 0000 0000 0000 0000 0000 0000 0100 x3000 0004
R4	
R29	
R30	
R31



- **问题：什么是JALR指令能够提供的而JAL指令无法提供的重要特点？**

问题：

```
Multiply:    ANDI    R2, R2, #0
LOOP :       BEQZ    R5, EXIT
              ADD     R2, R2, R4
              SUBI    R5, R5, #1
              J       LOOP
EXIT:        JR      R31
```

- 调用Multiply子例程的程序
 - 在执行JAL Multiply指令返回后，再次使用R5做运算，会发生什么情况？
 - R5：在子例程中已被改为0！

寄存器的保存/恢复

```
SaveR5:          .SPACE    #4
.....
.....
Multiply:        SW        SaveR5(r0), R5
                  ANDI      R2, R2, #0
LOOP :           BEQZ      R5, EXIT
                  ADD       R2, R2, R4
                  SUBI      R5, R5, #1
                  J         LOOP
EXIT:            LW        R5, SaveR5(r0)
                  JR        R31
```

- 采用callee-save（被调用者保存）策略

子例程的测试与调试

- 设置值
 - 寄存器、存储单元

Step命令

- 单独测试后，对整体进行测试和调试
- Step Into
 - 进入子例程内部并在子例程中单步执行
- Step Over
 - 不进入子例程内部，将子例程调用作为一步来执行
- Step Out
 - 进入子例程后，在认为不需要单步执行的时候，使用Step Out命令，一步完成子例程调用，返回到调用者

修改 (TRAP x09) 服务例程

- 子例程WriteChar
 - 注意：寄存器的保存与恢复

```
WriteChar:      sw      SaveR3(r0), r3
                 sw      SaveR5(r0), r5
                 lw      r5, DSR(r0)
L1:             lw      r3, 0(r5)
                 andi    r3, r3, #1
                 beqz    r3, L1
                 lw      r5, DDR(r0)
                 sw      0(r5), r2
                 lw      r3, SaveR3(r0)
                 lw      r5, SaveR5(r0)
                 jr      r31
```

子例程ReadChar

```
ReadChar:      sw      SaveR3(r0), r3
               sw      SaveR5(r0), r5
               lw      r5, KBSR(r0)
L3:            lw      r3, 0(r5)
               andi    r3, r3, #1
               beqz    r3, L3      ; 轮询直到一个字符被键入
               lw      r5, KBDR(r0)
               lw      r4, 0(r5)  ; 将输入的字符加载到R4
               lw      r3, SaveR3(r0)
               lw      r5, SaveR5(r0)
               jr      r31        ; 结束子例程
```

TRAP x09 服务例程-1

```

                                . data    x00003000
SaveR1                          . space   4                ; 保存寄存器的存储单元
SaveR2                          . space   4
SaveR3                          . space   4
SaveR5                          . space   4
SaveR31                         . space   4
.....
Newline                        . byte     x0A                ; 新行的ASCII码
.....

                                . text     x00003100
sw                             SaveR1(r0), r1                ; callee-save
sw                             SaveR2(r0), r2                ; callee-save
sw                             SaveR31(r0), r31              ; caller-save
```

TRAP x09 服务例程-2

;输出新行

lb r2, Newline(r0)

jal WriteChar

;输出提示符

addi r1, r0, Prompt ; 提示符字符串的起始地址

LOOP: lb r2, 0(r1) ; 输出提示符

beqz r2, Input ; 提示符字符串结束

jal WriteChar

addi r1, r1, #1 ; 提示符的指针加1

j LOOP ; 获取下一个提示符字符

TRAP x09 服务例程-3

;输入字符

Input: jal ReadChar

;把字符送给R2准备回显到显示器

addi r2, r4, #0

jal WriteChar

;输出新行

lb r2, Newline(r0)

jal WriteChar

TRAP x09 服务例程-4

```
lw      r1, SaveR1(r0) ; 将寄存器恢复为原先的值
lw      r2, SaveR2(r0)
lw      r31, SaveR31(r0)
jr      r31             ; 从TRAP返回
```

示例：两个多位整数加法

- 要点：通过键盘输入两个多位整数，且结果显示到显示器上

两个一位整数的加法

01	trap	x06	; 从键盘输入第一个数
02	trap	x07	; 回显
03	addi	r1, r4, #0	; 为下一次输入预留空间
04	trap	x06	; 从键盘输入第二个数
05	trap	x07	; 回显
06	add	r4, r1, r4	; 将两次输入相加
07	trap	x07	; 在显示器上显示结果

- 从键盘输入2
 - R4←字符“2”的ASCII码, x32
- 从键盘输入3
 - R4←字符“3”的ASCII码, x33
- $x32 + x33 \rightarrow x65$
 - 字母“e”的ASCII码

数据类型转换

- (1) 输入：
 - ASCII码 → 二进制补码整数
- (2) 输出：
 - 二进制补码整数 → ASCII码

考虑数据类型转换的一位数加法

01	trap	x06	; 从键盘输入第一个字符
02	trap	x07	; 回显
03	subi	r1, r4, x30	; R1得到第一个整数
04	trap	x06	; 从键盘输入第二个字符
05	trap	x07	; 回显
06	subi	r4, r4, x30	; R4得到第二个整数
07	add	r4, r1, r4	; 将两次输入相加
08	addi	r4, r4, x30	; R4得到加法结果的ASCII码
09	trap	x07	; 在显示器上显示结果

- 1位整数

- ASCII码 → 二进制补码整数

- 减x30

- 二进制补码整数 → ASCII码

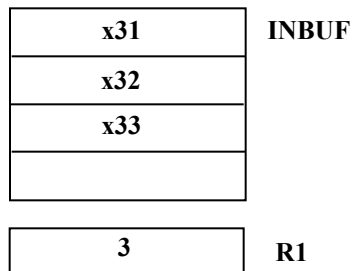
- 加x30

多位整数

- 通过键盘输入一个多位整数
 - ASCII码字符串 → 二进制补码整数
- 在显示器上显示一个多位整数
 - 二进制补码整数 → ASCII码字符串

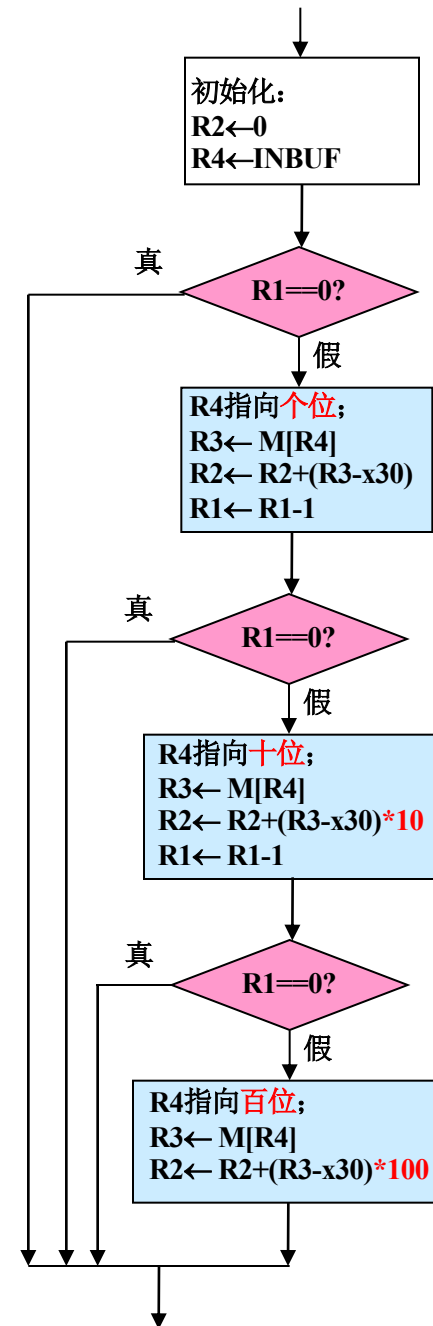
子例程: ASCIIToInt

- 正的十进制数的ASCII码字符串 → 二进制补码整数
 - R1, 位数
 - 如: "123"



流程图：ASCIIToInt

- R2: int, 正整数
- R1: 位数
- R4: 指针
- 从个位开始转换
 - 注: 最多3位正整数
 - 改进?



子例程：ASCIIToInt--1

```
;
; 将一个最多3位的正十进制数的ASCII码字符串转换成二进制整数。
; INBUF中存储的是这个ASCII码字符串中的最高位；
; R1还有多少位数未被处理；R2用来存储结果。
;

      data
.....
LookUp10: .word      0, 10, 20, 30, 40, 50, 60, 70, 80, 90
LookUp100: .word     0, 100, 200, 300, 400, 500, 600, 700, 800, 900
SaveR1:    .space    4
SaveR3:    .space    4
.....

      .text

;
ASCIIToInt: sw      SaveR1(r0), r1      ;保存寄存器
            sw      SaveR3(r0), r3
            sw      SaveR4(r0), r4
            sw      SaveR5(r0), r5

; 初始化

            addi    r4, r0, INBUF
            addi    r2, r0, #0          ; R2用于存储结果
            beqz    r1, DoneAtoB       ; 没有位数需要处理

; 个位

            add     r4, r4, r1
            subi    r4, r4, #1          ; R4指向个位
            lb      r3, 0(r4)          ; R3, 个位
            subi    r3, r3, x30         ; 减去ASCII码模板
            add     r2, r2, r3          ; 加上个位数的贡献
```

```
;

            subi    r1, r1, #1
            beqz    r1, DoneAtoI       ; 原数字只有一位

; 十位，查表计算

            subi    r4, r4, #1          ; R4现在指向十位
            lb      r3, 0(r4)          ; R3, 十位
            subi    r3, r3, x30         ; 减去ASCII码模板
            addi    r5, r0, LookUp10    ; LookUp10是十位
                                           ; 数的基数
            slli    r3, r3, #2          ; R3<-R3*4
            add     r5, r5, r3          ; R5指向真正的十位
                                           ; 数的值

            lw      r3, 0(r5)
            add     r2, r2, r3          ; 将十位数的贡献加
                                           ; 入总和

;

            subi    r1, r1, #1
            beqz    r1, DoneAtoI       ; 原数只有两位
```

子例程: ASCIIToInt--2

; 百位

```
subi    r4, r4, #1           ; R4现在指向百位
lb      r3, 0(r4)            ; R3, 百位
subi    r3, r3, x30          ; 减去ASCII码模板
addi    r5, r0, LookUp100    ; LookUp100是百位数的基数
slli    r3, r3, #2           ; R3<-R3*4
add     r5, r5, r3           ; R5指向真正的百位数的值
lw      r3, 0(r5)
add     r2, r2, r3           ; 将百位数的贡献加入总和
```

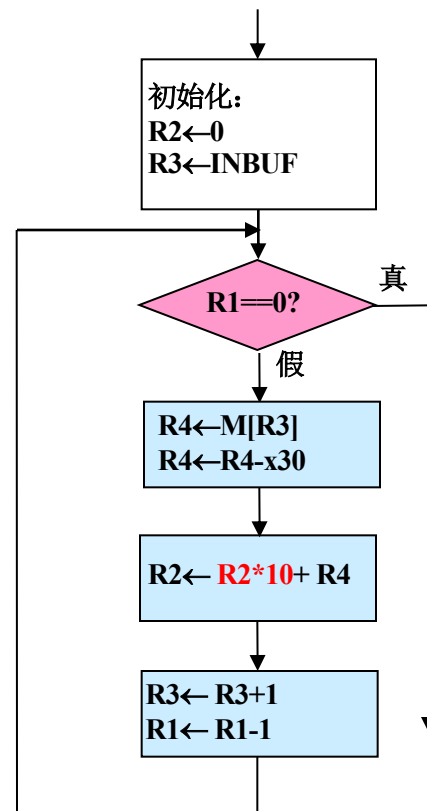
;

;恢复寄存器

```
DoneAtol: lw      r1, SaveR1(r0)
           lw      r3, SaveR3(r0)
           lw      r4, SaveR4(r0)
           lw      r5, SaveR5(r0)
           jr      r31
```


流程图2: ASCIIToInt

- 从最高位开始转换
- R2: int, 正整数
- R1: 位数
- R3: 指针



子例程: ASCIIToInt

```
;  
; 这个算法将一个正的十进制数的ASCII码字符串转换成二进制整数。  
; ASCII码字符串被存储在从INBUF开始的存储单元中; R1记录位数。  
; R2用来存储结果。  
;
```

```
        .data  
INBUF:   .space    20
```

```
.....
```

```
SaveR1:  .space    4  
SaveR2:  .space    4  
SaveR3:  .space    4  
SaveR4:  .space    4  
SaveR5:  .space    4  
SaveR6:  .space    4
```

```
;
```

```
        .text
```

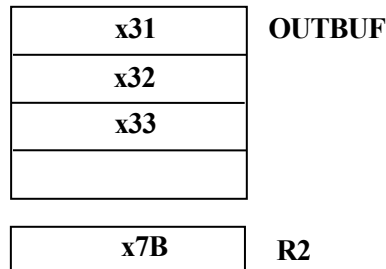
```
;
```

```
ASCIIToInt:  sw      SaveR1(r0), r1 ;保存寄存器  
             sw      SaveR3(r0), r3  
             sw      SaveR4(r0), r4  
             sw      SaveR5(r0), r5  
             sw      SaveR6(r0), r6
```

```
;  
        addi      r2, r0, #0      ; R2用于存储结果  
        addi      r3, r0, INBUF   ; R3指向ASCII码字符串  
;  
AToILoop: beqz      r1, DoneAtol    ; 没有位数需要处理  
        lb        r4, 0(r3)       ; 从最高位开始依次取出ASCII码  
        subi      r4, r4, x30     ; 减去ASCII码模板  
;  
        addi      r6, r2, #0      ; R6<- R2*10  
        addi      r5, r0, #9  
Mult10:  beqz      r5, Next  
        add       r6, r6, r2  
        subi      r5, r5, #1  
        j         Mult10  
Next:    add       r2, r6, r4      ; R2<-R2*10+R4  
;  
        addi      r3, r3, #1      ; R3指向下一个字符  
        subi      r1, r1, #1      ; 剩余位数  
        j         AToILoop  
;  
DoneAtol: lw        r1, SaveR1(r0) ; 恢复寄存器  
        lw        r3, SaveR3(r0)  
        lw        r4, SaveR4(r0)  
        lw        r5, SaveR5(r0)  
        lw        r6, SaveR6(r0)  
        jr        r31
```

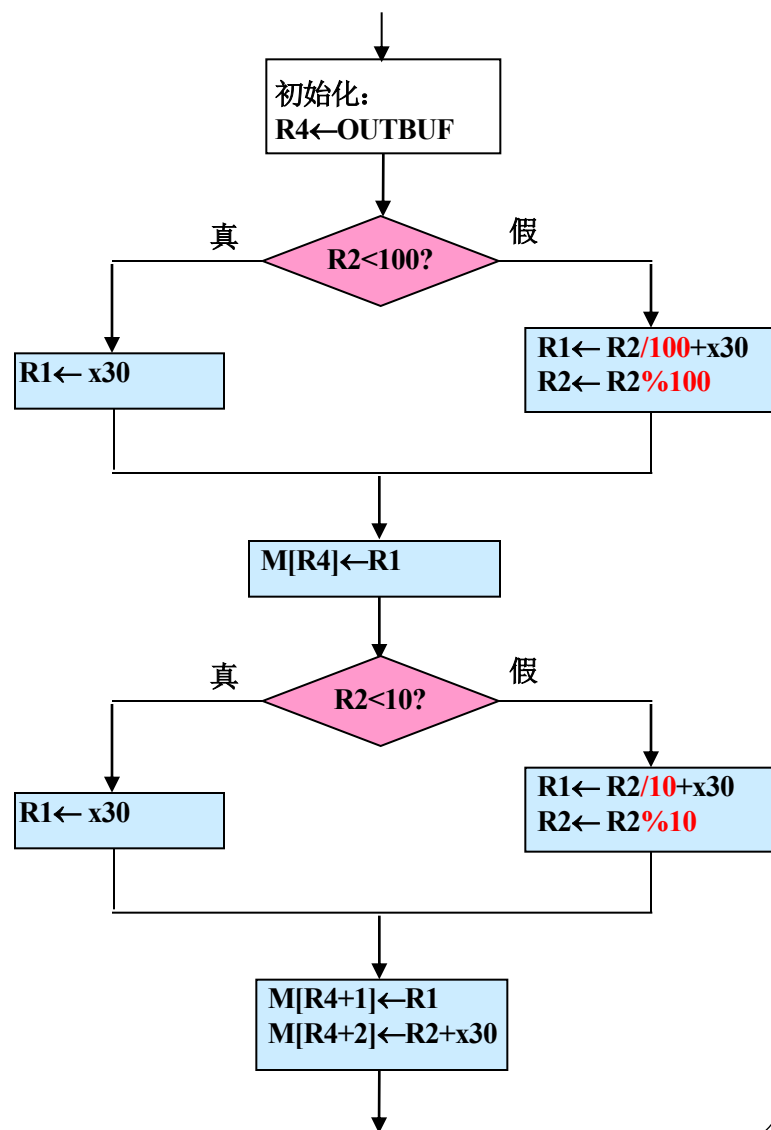
子例程: **IntToASCII**

- 正的二进制补码整数 → 十进制数的ASCII码字符串
 - R2, 二进制补码整数



流程图: IntToASCII

- R4: 指针
- 注: R2中的数值范围为0~+999
 - 从百位开始转换
 - 如果R2的值为23, 则存储 “023”
 - 改进?



子例程:

IntToASCII

(1)

```
;
; 这个算法将一个范围在0到+999之间的二进制补码整数转换为一个由
; 3个十进制数位组成的ASCII码字符串。
; R2里包含的是最初要转换的数。
;
IntToASCII:    sw          SaveR1(r0), r1 ; 保存寄存器
               sw          SaveR2(r0), r2
               sw          SaveR3(r0), r3
               sw          SaveR4(r0), r4

; 初始化
               addi        r4, r0, OUTBUF; R4指向即将生成的字符串

; 判断百位数
               addi        r1, r0, x30 ; 为百位数准备
Loop100:       slti        r3, r2, #100
               bnez        r3, End100
               addi        r1, r1, #1 ; R2/100
               subi        r2, r2, #100
               j           Loop100

;
End100:        sb          0(r4), r1 ; 存储百位数的ASCII码
```

子例程:

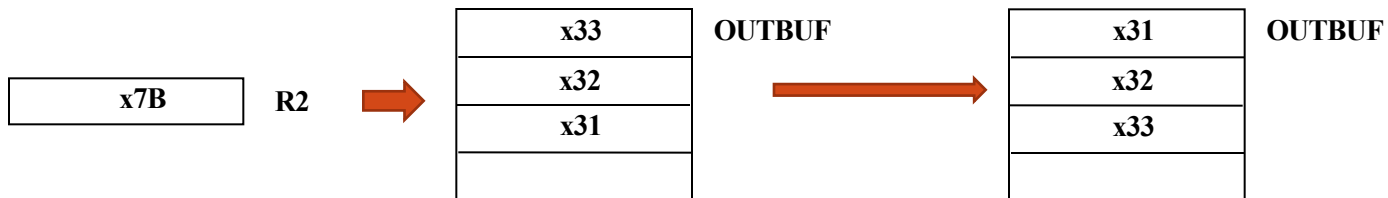
IntToASCII

(2)

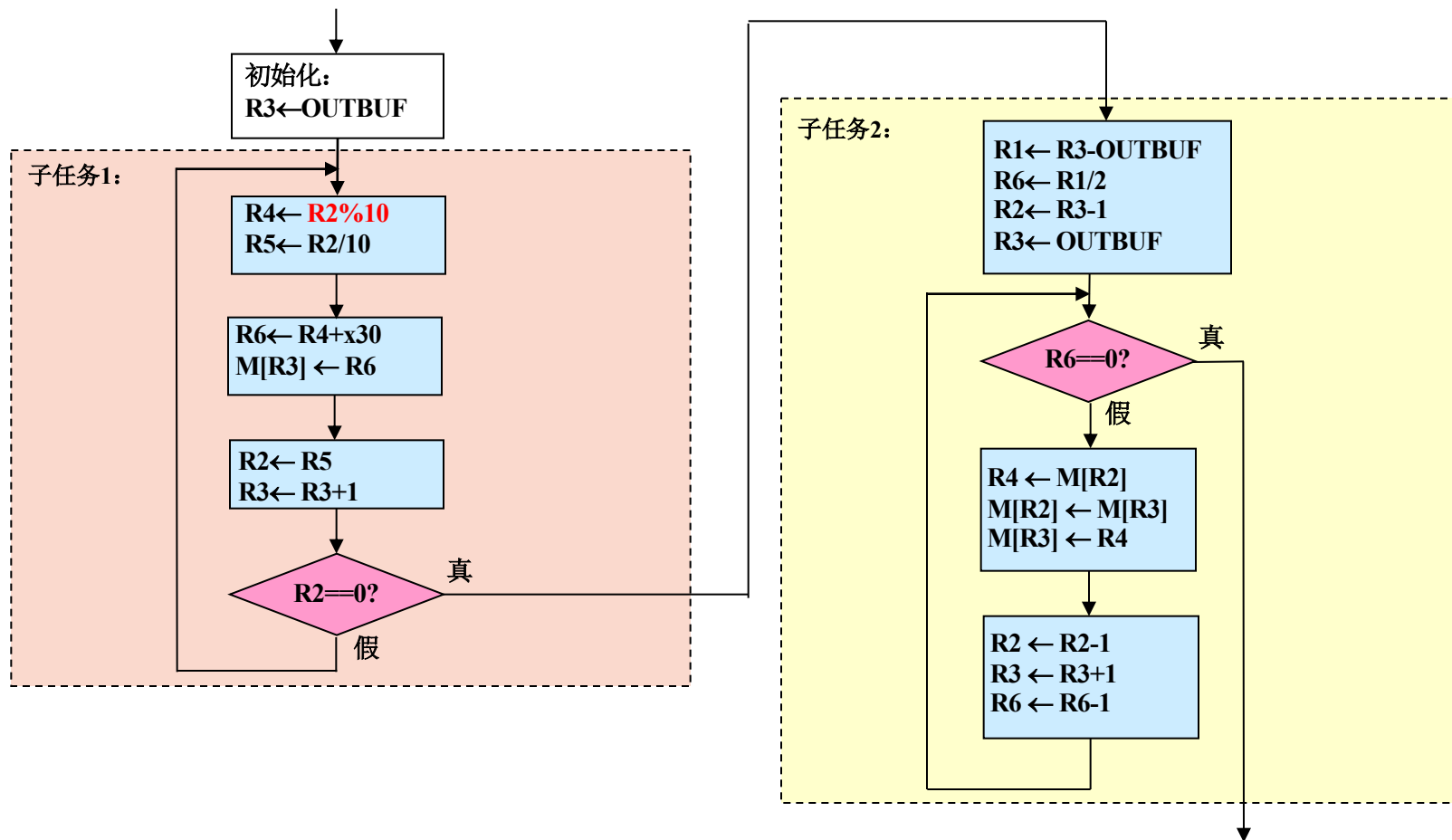
```
; 判断十位数
                                addi      r1, r0, x30 ; 为十位数准备
Loop10:                        slti      r3, r2, #10
                                bnez      r3, End10
                                addi      r1, r1, #1  ; R2/10
                                subi      r2, r2, #10
                                j          Loop10
;
End10:                          sb        1(r4), r1  ; 存储十位数的ASCII码
; 个位数
                                addi      r1, r2, x30 ; 个位数
                                sb        2(r4), r1
;
                                lw         r1, SaveR1(r0) ; 恢复寄存器
                                lw         r2, SaveR2(r0)
                                lw         r3, SaveR3(r0)
                                lw         r4, SaveR4(r0)
                                jr         r31
```

子例程： **IntToASCII** ——2

- R3: 指针
- R1: 记录位数
- 注: **R2**中的数值为正数
 - 从个位开始转换
 - 1、先, 按照从个位到最高位的顺序存储字符串
 - 2、再, 字符串逆序



流程图2: IntToASCII



子例程:

IntToASCII

(1)

```
;
; 这个算法将一个正的二进制补码整数转换为一个ASCII码字符串。
; R2里包含的是最初要转换的数。
; 转换后的ASCII码字符串被存储在从OUTBUF开始的存储单元中; R1记录整数位数。
;
IntToASCII:    sw          SaveR2(r0), r2          ; 保存寄存器
               sw          SaveR3(r0), r3
               sw          SaveR4(r0), r4
               sw          SaveR5(r0), r5
               sw          SaveR6(r0), r6

;
               addi         r3, r0, OUTBUF          ; R3指向起始单元
;
; 子任务1, 将二进制数转换为ASCII码字符串, 按照从个位到最高位的顺序存储
;
STask1:        addi         r4, r2, #0              ; R4 <- R2 % 10
               addi         r5, r0, #0              ; R5 <- R2 / 10
Divid10 :      slti         r6, r4, #10
               bnez         r6, STExit
               subi         r4, r4, #10
               addi         r5, r5, #1
               j            Divid10
STExit :       addi         r6, r4, x30              ; 余数加上ASCII码模板
               sb           0(r3), r6               ; 存储
               addi         r2, r5, #0              ; R2 <- R2 / 10
               addi         r3, r3, #1              ; R3指向下一个单元
               beqz         r2, STask2               ; 没有位数需要处理
               j            STask1
```

子例程:

IntToASCII (2)

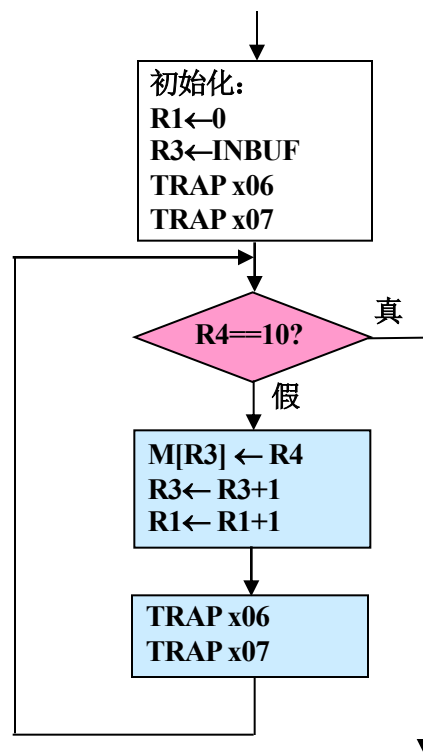
```
;
; 子任务2, 字符串逆序
;
STask2 :  subi    r1, r3, OUTBUF          ; R1记录位数
          srli    r6, r1, #1              ; R6<-R1/2, 交换次数
          subi    r2, r3, #1              ; R2 指向最后一个字符
          addi    r3, r0, OUTBUF          ; R3 指向第一个字符
ST2Loop : beqz    r6, DoneItoA            ; 交换字符
          lb      r4, 0(r2)
          lb      r5, 0(r3)
          sb      0(r2), r5
          sb      0(r3), r4
          subi    r2, r2, #1
          addi    r3, r3, #1
          subi    r6, r6, #1
          j       ST2Loop
;
DoneItoA: lw      r2, SaveR2(r0)           ; 恢复寄存器
          lw      r3, SaveR3(r0)
          lw      r4, SaveR4(r0)
          lw      r5, SaveR5(r0)
          lw      r6, SaveR6(r0)
          jr      r31
```

子例程: InputASCII

- 通过键盘输入一个多位十进制正整数
 - 存储到INBUF开始的存储单元中
 - 使用R1记录位数

流程图：InputASCII

- R3：指针
- R1：正整数的位数
- 回车结束



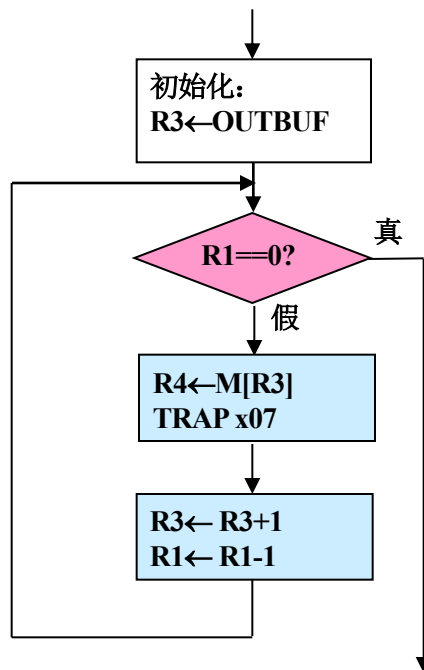
子例程: InputASCII

```
; 输入一个十进制正整数。
; ASCII码字符串被存储在从INBUF开始的存储单元中。
; R1记录位数。
InputASCII:  sw      SaveR3(r0), r3          ; 保存寄存器
            sw      SaveR4(r0), r4
            sw      SaveR5(r0), r5
            sw      SaveR31(r0), r31       ; caller-save
;
            addi    r3, r0, INBUF          ; R3指向起始单元
            addi    r1, r0, #0            ; R1记录位数
;
INLoop:     trap    x06                   ; 输入字符
            trap    x07                   ; 回显
            seqi    r5, r4, x0A           ; 测试是否输入回车
            bnez    r5, DoneInput
            sb      0(r3), r4             ; 存储最近读入的字符
            addi    r3, r3, #1
            addi    r1, r1, #1
            j       INLoop
;
DoneInput:  lw      r3, SaveR3(r0)         ; 恢复寄存器
            lw      r4, SaveR4(r0)
            lw      r5, SaveR5(r0)
            lw      r31, SaveR31(r0)
            jr      r31
```

子例程: **OutputASCII**

- 将存储在OUTBUF开始的一段存储单元中的字符串，在显示器上逐一显示出来
 - 字符个数, R1

流程图：OutputASCII



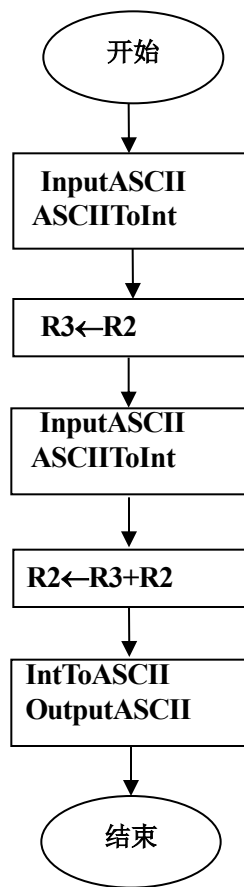
子例程：OutputASCII

```
;
; 输出一个字符串。
; ASCII码字符串被存储在从OUTBUF开始的存储单元中；R1记录位数。
;
OutputASCII:  sw          SaveR1(r0), r1          ; 保存寄存器
              sw          SaveR3(r0), r3
              sw          SaveR4(r0), r4
              sw          SaveR31(r0), r31

;
              addi        r3, r0, OUTBUF        ; R3指向起始单元
;
OutLoop:     beqz         r1, DoneOutput
              lb          r4, 0(r3)             ; 依次取出字符
              trap        x07                   ; 显示输出
              addi        r3, r3, #1
              subi        r1, r1, #1
              j           OutLoop

;
DoneOutput:  lw          r1, SaveR1(r0)          ; 恢复寄存器
              lw          r3, SaveR3(r0)
              lw          r4, SaveR4(r0)
              lw          r31, SaveR31(r0)
              jr          r31
```


两个多位整数加法



main

```
;  
; 取得用户输入的两个正的十进制数的ASCII码序列,  
; 通过调用ASCIIToInt子例程, 将其转换为两个  
; 二进制整数, 执行加法运算,  
; 将结果调用IntToASCII子例程, 转换为ASCII码  
; 序列, 并显示在显示器上。
```

```
        .data  
PromptMsg1: .asciiz    "Enter the first number: "  
PromptMsg2: .asciiz    "Enter the second number: "  
PromptMsg3: .asciiz    "The sum of two numbers is: "  
  
        .align        2  
INBUF:    .space       20  
OUTBUF:   .space       20  
SaveR1:   .space       4  
SaveR2:   .space       4  
SaveR3:   .space       4  
SaveR4:   .space       4  
SaveR5:   .space       4  
SaveR6:   .space       4  
SaveR31:  .space       4  
;
```

```
        .text  
        .global      main  
main:   addi          r4, r0, PromptMsg1  
        trap         x08                ;输出提示符  
        jal          InputASCII  
        jal          ASCIIToInt  
; 得到第一个整数  
        addi          r3, r2, #0  
; 输入第二个整数  
        addi          r4, r0, PromptMsg2  
        trap         x08  
        jal          InputASCII  
        jal          ASCIIToInt  
; 执行加法运算  
        add           r2, r3, r2  
; 转换为ASCII字符串  
        jal          IntToASCII  
; 输出结果  
        addi          r4, r0, PromptMsg3  
        trap         x08  
        jal          OutputASCII  
        trap         x00
```

传给服务例程/子例程的值

- 字符输出 (TRAP x07) 服务例程
 - 字符, R4
- 字符串输出 (TRAP x08) 服务例程
 - 字符串起始地址, R4
- InputASCII
 - 起始地址, INBUF
- ASCIIToInt
 - 位数, R1
 - INBUF
- IntToASCII
 - 整数, R2
 - OUTBUF
- OutputASCII
 - 起始地址, OUTBUF
 - 位数, R1

返回值

- **Return Values**
- 从服务例程/子例程传出的值
- 字符输入 (TRAP x06) 服务例程
 - 字符, R4
- InputASCII
 - 位数, R1
- ASCIIToInt
 - 整数, R2
- IntToASCII
 - 位数, R1

使用子例程

- 必须知道
 - 地址(或标记)
 - 功能
 - 不需要知道如何实现
 - 传给子例程的值
 - 返回值

寄存器的保存和恢复

- 通常使用被调用者保存（callee-save）策略，返回值除外
- 注意：R31的保存

库例程

- 库：一个已测试的组件的集合
 - 模块化设计
 - 不需要了解其内部细节
 - 例如，平方根（SQRT）

计算直角三角形斜边长

```
01          .data
02 Side1:    .space 4
03 Side2:    .space 4
04 Hypot:    .space 4
05          .....
06          .text
07          .....
06          lw      r1, Side1(r0)
07          jal      SQUARE
08          addi     r2, r1, #0
09          lw      r1, Side2(r0)
0A          jal      SQUARE
0B          add     r1, r1, r2
```


jal Sqrt

0C		jal	Sqrt	
0D		sw	Hypot (r0), r1	
0E		J	NEXT_TASK	
0F				
10	SQUARE:		;寄存器R3, R4的保存
11	addi	r3, r1, #0		
12		addi	r4, r1, #0	
13	AGAIN:	subi	r3, r3, #1	
14		beqz	r3, DONE	
15		add	r1, r1, r4	
16		j	AGAIN	
			;寄存器R3, R4的恢复
17	DONE:	jr	r31	
18	;			
19	Sqrt:		; R1←Sqrt (R1)
...	...			; 如何写这个子例程?
1A		jr	r31	

数学库

- 数学库提供了许多子例程（包括SQRT）
 - 用户程序员需要知道的是实现平方根功能的库例程的目标地址，在哪里输入数值 x ，在哪里可以得到结果
 - 可以非常方便的从数学库附带的文档中获得

SQRT

- 如果库程序起始于地址SQRT，提供给库程序的数值x在R1中，库程序的结果也在R1中

```
01          . data
02 Side1:    . space 4
03 Side2:    . space 4
04 Hypot:    . space 4
05          .....
06          . text
07          . extern      SQRT
08          .....
```

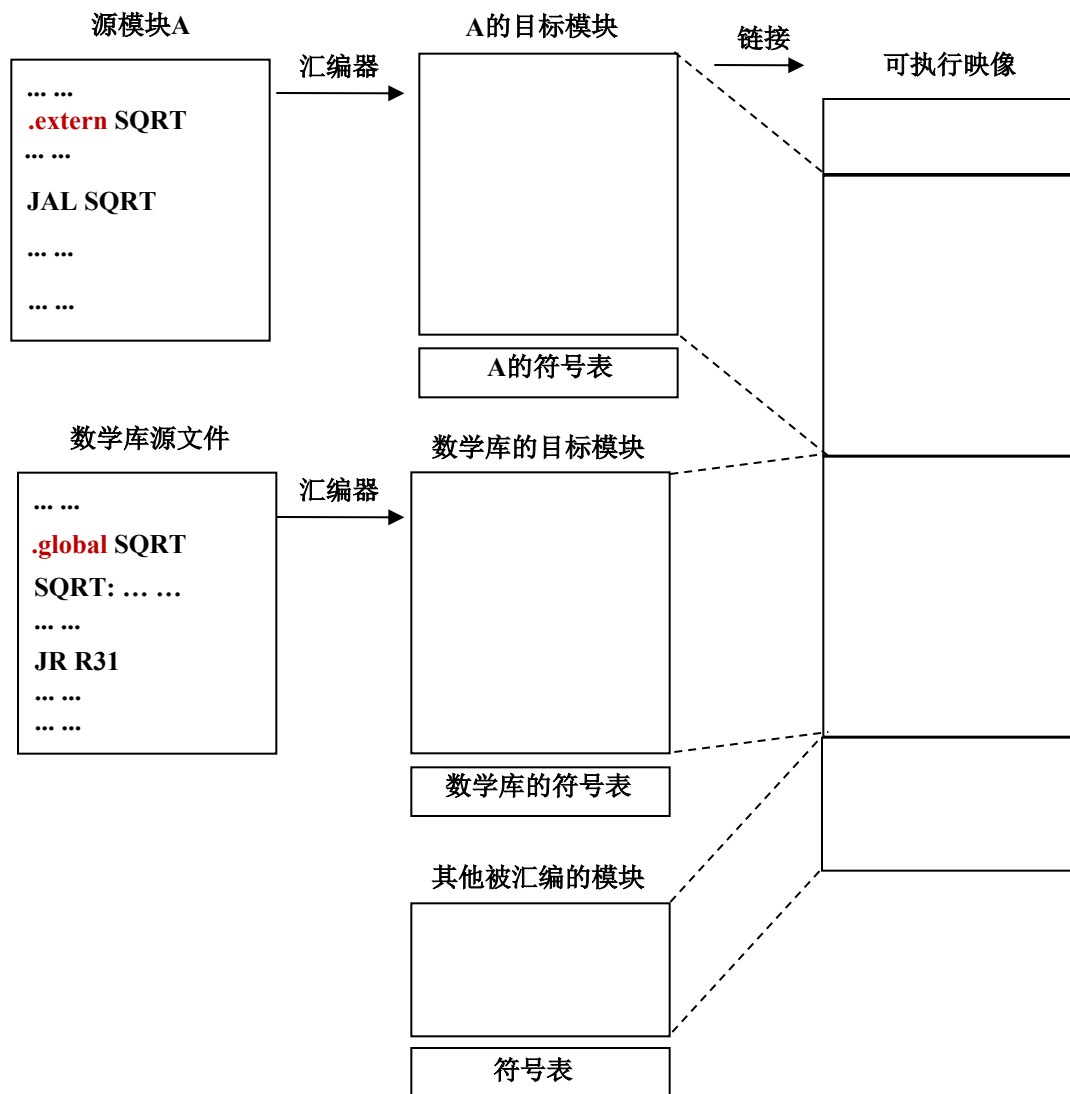
jal SQRT

07	lw	r1, Side1(r0)
08	jal	SQUARE
09	addi	r2, r1, #0
0A	lw	r1, Side2(r0)
0B	jal	SQUARE
0C	add	r1, r1, r2
0D	jal	SQRT
0E	sw	Hypot (r0), r1
0F	J	NEXT_TASK
10		

SQUARE

```
11 SQUARE:      .....                ;寄存器R3, R4的保存
12              addi    r3, r1, #0
13              addi    r4, r1, #0
14 AGAIN:        subi    r3, r3, #1
15              beqz    r3, DONE
16              add     r1, r1, r4
17              j       AGAIN
18              .....                ;寄存器R3, R4的恢复
19 DONE:         jr      r31
20              ... ..
```

由多个文件构建可执行映像



C的标准库

- 许多通用功能
 - 例如：I/O，字符串处理，数学函数等
- 一般由编译器和操作系统的设计者提供

习题

- 上机

- 14. 2

- 14. 3

- 14. 4

- 3)

- 14. 5

- 1)

- 书面作业

- 14. 7

- 14. 8

- 14. 9

- 14. 10

递归子例程

- 调用它本身的子例程
- 例如，计算 $n!$
 - 可以采用循环结构，也可以采用递归方程

$$f(n) = n * f(n-1)$$

$$f(1) = 1$$

示例：计算 $n!$

- 递归方程

$$\text{Factorial}(n) = n * \text{Factorial}(n-1)$$

- 初始条件

$$\text{Factorial}(1) = 1$$

- $n=4$

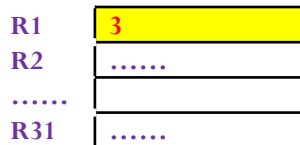
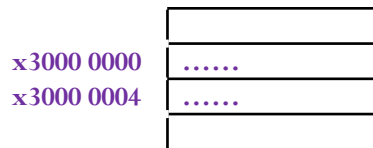
$$\begin{aligned}\text{Factorial}(4) &= 4 * \text{Factorial}(3) \\ &= 4 * 3 * \text{Factorial}(2) \\ &= 4 * 3 * 2 * \text{Factorial}(1) \\ &= 4 * 3 * 2 * 1\end{aligned}$$

递归子例程 FACTORIAL

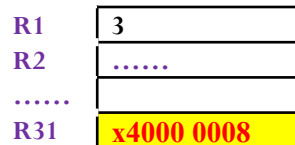
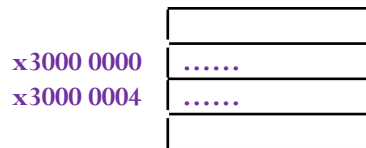
```
09      FACTORIAL: sw      SaveR1(r0), r1
0A                      sw      SaveR31(r0), r31
0B                      seqi     r5, r1, #1           ;n=1?
0C                      bnez     r5, EXIT1
0D                      subi     r1, r1, #1           ;n-1
0E                      jal      FACTORIAL            ;f(n-1)
0F                      addi     r3, r2, #0           ;R3 ← f(n-1)
10                      addi     r4, r1, #1           ;R4 ← n
11                      andi     r2, r2, #0           ; R2 ← 0
12      LOOP :      beqz     r4, EXIT2                ;计算R2 ← n * f(n-1)
13                      add      r2, r2, r3
14                      subi     r4, r4, #1
15                      j        LOOP
16      EXIT1 :      addi     r2, r0, #1               ;f(1)=1
17      EXIT2 :      lw      r1, SaveR1(r0)
18                      lw      r31, SaveR31(r0)
19                      jr      r31
```

n=3

01	.data	x30000000	
02 SaveR1 :	.space	4	
03 SaveR31 :	.space	4	
04	.text	x40000000	
05	.global	main	
06 main :	addi	r1, r0, #3	; n=3
07	jal	FACTORIAL	; f(n)
08	trap	x00	
.....			



(1) 执行07行前



(2) 执行07行后，计算f(3)

.....			
09 FACTORIAL:	sw	SaveR1(r0), r1	
0A	sw	SaveR31(r0), r31	
0B	seqi	r5, r1, #1	;n==1?
0C	bnez	r5, EXIT1	
0D	subi	r1, r1, #1	;n-1
0E	jal	FACTORIAL	;f(n-1)
.....			

x3000 0000	3
x3000 0004	x4000 0008

R1	3
R2
.....	
R31	x4000 0008

(3) 执行0A行后

x3000 0000	3
x3000 0004	x4000 0008

R1	2
R2
.....	
R31	x4000 0008

(4) 执行0E行前

x3000 0000	3
x3000 0004	x4000 0008

R1	2
R2
.....	
R31	x4000 0024

(5) 执行0E行后,计算f(2)

.....			
09 FACTORIAL:	sw	SaveR1(r0), r1	
0A	sw	SaveR31(r0), r31	
0B	seqi	r5, r1, #1	;n==1?
0C	bnez	r5, EXIT1	
0D	subi	r1, r1, #1	;n-1
0E	jal	FACTORIAL	;f(n-1)
.....			

x3000 0000	2
x3000 0004	x4000 0024

R1	2
R2
.....	
R31	x4000 0024

(6) 执行0A行后

x3000 0000	2
x3000 0004	x4000 0024

R1	1
R2
.....	
R31	x4000 0024

(7) 执行0E行前

x3000 0000	2
x3000 0004	x4000 0024

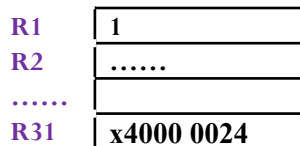
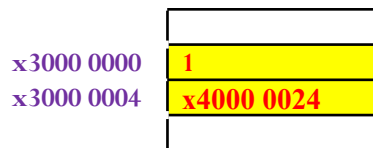
R1	1
R2
.....	
R31	x4000 0024

(8) 执行0E行后,计算f(1)

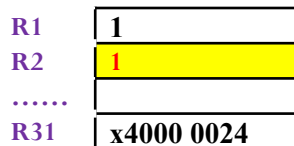
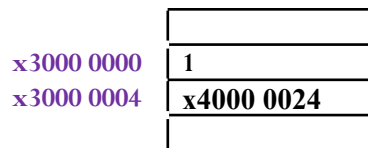
```

.....
09 FACTORIAL:      sw      SaveR1(r0), r1
0A                sw      SaveR31(r0), r31
0B                seqi     r5, r1, #1          ;n==1?
0C                bnez     r5, EXIT1
0D                subi     r1, r1, #1          ;n-1
0E                jal      FACTORIAL          ;f(n-1)
.....
16 EXIT1 :         addi     r2, r0, #1          ;f(1)=1
17 EXIT2 :         lw       r1, SaveR1(r0)
18                lw       r31, SaveR31(r0)
19                jr       r31

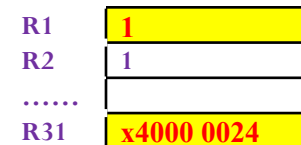
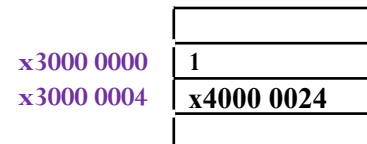
```



(9) 执行0A行后



(10) 执行17行前
f(1)=1



(11) 执行18行后

```

.....
0E      jal      FACTORIAL      ;f(n-1)
0F      addi     r3, r2, #0      ;f(n-1)
10      addi     r4, r1, #1      ;n
11      andi     r2, r2, #0
12 LOOP: beqz     r4, EXIT2      ;计算n * f(n-1)
13      add      r2, r2, r3
14      subi     r4, r4, #1
15      j        LOOP
16 EXIT1: addi     r2, r0, #1      ;f(1)=1
17 EXIT2: lw      r1, SaveR1(r0)
18      lw      r31, SaveR31(r0)
19      jr      r31

```

x3000 0000	1
x3000 0004	x4000 0024

R1	1
R2	1
R3	1
R4	2
.....	
R31	x4000 0024

(12) 执行11行前
n=2, f(n-1)=1

x3000 0000	1
x3000 0004	x4000 0024

R1	1
R2	2
R3	1
R4	0
.....	
R31	x4000 0024

(13) 执行17行前
计算2*f(1), 即f(2)

x3000 0000	1
x3000 0004	x4000 0024

R1	1
R2	2
R3	1
R4	0
.....	
R31	x4000 0024

(14) 执行18行后


```

.....
0E      jal      FACTORIAL      ;f(n-1)
0F      addi     r3, r2, #0      ;f(n-1)
10      addi     r4, r1, #1      ;n
11      andi     r2, r2, #0
12 LOOP: beqz     r4, EXIT2      ;计算n * f(n-1)
13      add      r2, r2, r3
14      subi     r4, r4, #1
15      j        LOOP
16 EXIT1: addi     r2, r0, #1      ;f(1)=1
17 EXIT2: lw      r1, SaveR1(r0)
18      lw      r31, SaveR31(r0)
19      jr       r31

```

x3000 0000	1
x3000 0004	x4000 0024

R1	1
R2	2
R3	2
R4	2
.....	
R31	x4000 0024

(15) 执行11行前
应计算3*f(2)，无法得到3，出错！

“栈” 机制

- 造成错误的原因
 - 递归调用子例程时，保存寄存器的指令将前一次保存的值覆盖了
 - 如何解决？
- 答案
 - 采用 “栈” 机制

栈 —— 一种抽象数据类型

- 栈是一种存储结构
 - 可通过不同的方式实现
- 栈的概念
 - 与实现无关
 - 后进先出 (Last In, First Out), LIFO
- 抽象数据类型
 - 存储机制, 由对它执行的操作所定义, 而不是实现它的方式

栈 —— 示例

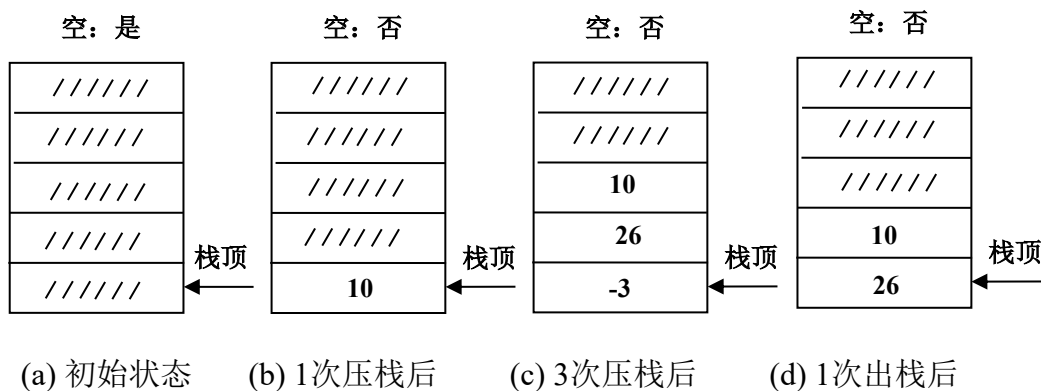
- 洗盘子
 - 每洗净一个盘子，就放到另一个已经洗好的盘子上面；
 - 取盘子时，则是从这摞盘子中一个接一个的向下拿。
 - 后进先出
 - 最后摆放上去的盘子是最先要拿走的

PUSH/POP

- **压栈** (push)
 - 把一个元素插入栈
- **出栈** (pop)
 - 移出一个元素

硬件栈

- 由一定数量的寄存器组成，每个寄存器可以存储一个元素
- 当每个元素被存入或取出时，已经在栈里的**元素会移动**（方向与洗盘子的例子相反）



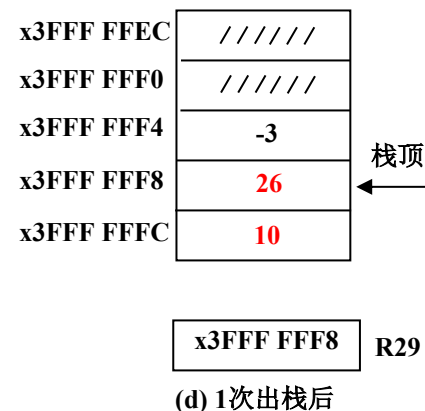
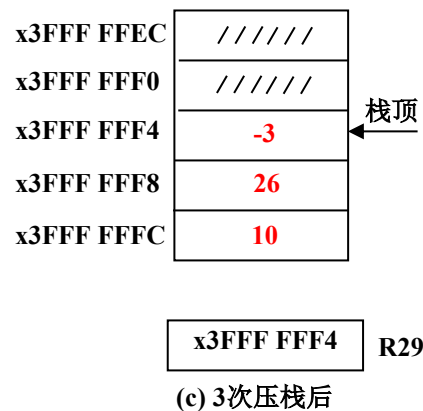
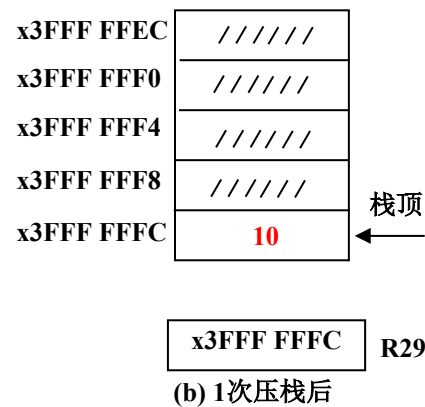
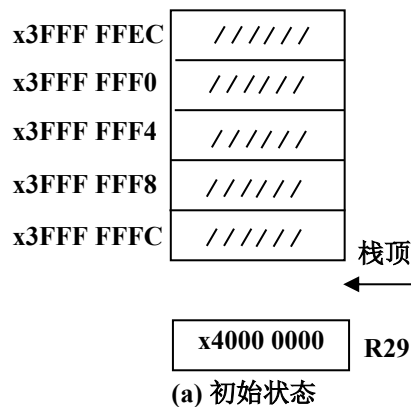
在存储器中实现栈

- 由一组存储单元和被称为“**栈指针**”的机制组成

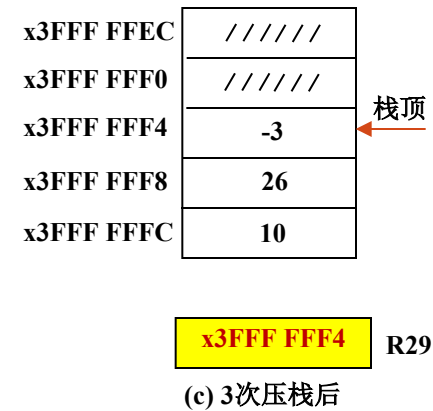
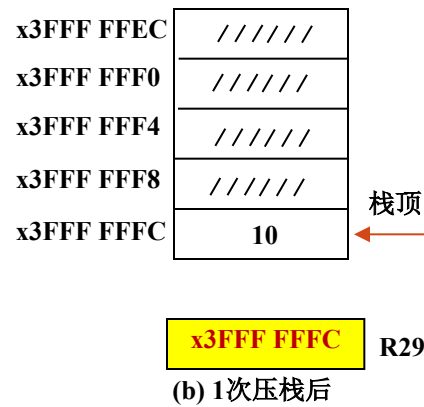
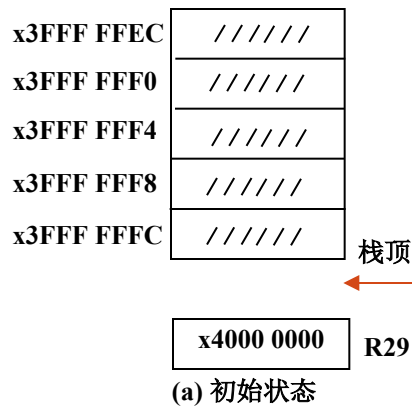
- 栈指针

- 栈的栈顶
- 最后压入的元素的存储单元地址

- 栈中的数据不进行物理移动

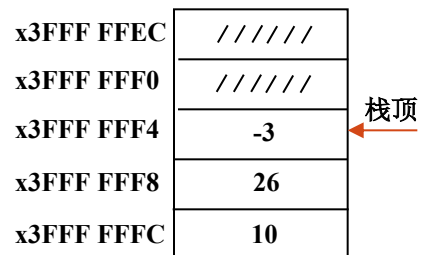


PUSH



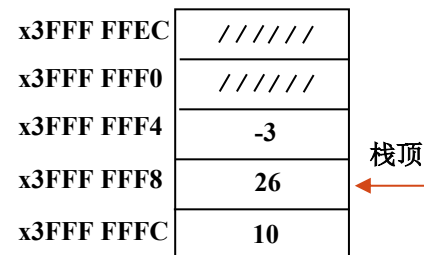
push: **subi** **r29, r29, #4**
 sw **0(r29), r2**

POP



x3FFF FFF4 R29

(c) 3次压栈后



x3FFF FFF8 R29

(d) 1次出栈后

```
pop:      lw      r2, 0(r29)
          addi     r29, r29, #4
```

控制存储器访问的栈机制

- “**栈协议**”，控制访问规则
 - “栈”要求：
 - 通过执行PUSH指令序列实现压栈
 - 通过执行POP指令序列实现出栈
 - 后进先出

```
push:          subi          r29, r29, #4
               sw             0(r29), r2

pop:           lw             r2, 0(r29)
               addi          r29, r29, #4
```

采用栈机制的n!

```
01      .data      x30000000
02 STACK : .space   40
03;
04      .text      x40000000
05      .global    main
06 main :  addi     r29, r0, STACK
07          addi     r29, r29, #40
08          addi     r1, r0, #3      ;n=3
09          jal      FACTORIAL      ;f(n)
0A          trap     x00
```

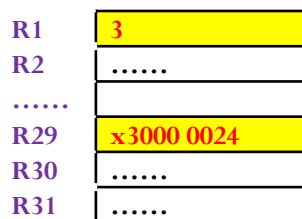
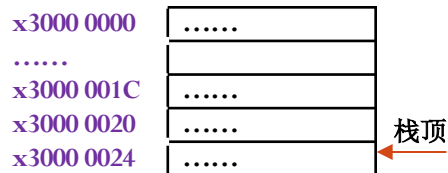
```
0B FACTORIAL:subi     r29, r29, #4
0C          sw       0(r29), r1
0D          subi     r29, r29, #4
0E          sw       0(r29), r31
0F          seqi     r5, r1, #1    ;n=1?
10          bnez     r5, EXIT1
11          subi     r1, r1, #1    ;n--
12          jal      FACTORIAL    ;f(n-1)
13          addi     r3, r2, #0    ; f(n-1)
14          addi     r4, r1, #1    ;n
15          andi     r2, r2, #0
16 LOOP :   beqz     r4, EXIT2    ;计算n * f(n-1)
17          add      r2, r2, r3
18          subi     r4, r4, #1
19          j        LOOP
1A EXIT1 :   addi     r2, r0, #1    ;f(1)=1
1B EXIT2 :   lw       r31, 0(r29)
1C          addi     r29, r29, #4
1D          lw       r1, 0(r29)
1E          addi     r29, r29, #4
1F          jr       r31
```

栈

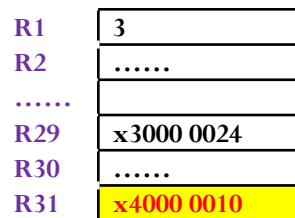
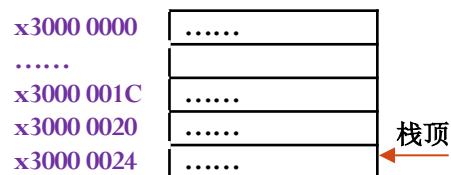
```

01          .data      x30000000
02 STACK :   .space    40
03;
04          .text      x40000000
05          .global    main
06 main :    addi       r29, r0, STACK
07          addi       r29, r29, #40
08          addi       r1, r0, #3      ;n=3
09          jal        FACTORIAL    ;f(n)
0A          trap       x00
.....

```



(1) 执行09行前

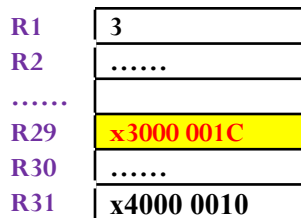
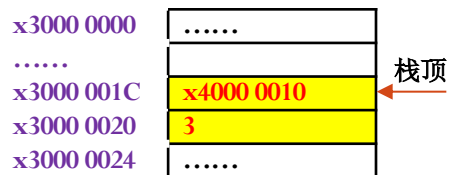


(2) 执行09行后

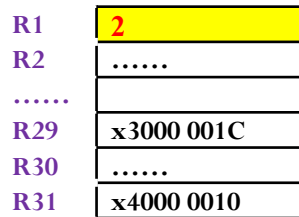
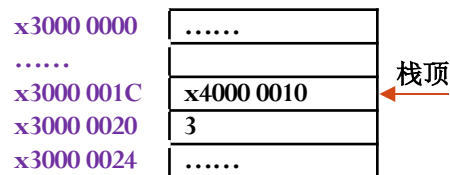
```

0B FACTORIAL:subi      r29, r29, #4
0C          sw          0(r29), r1
0D          subi       r29, r29, #4
0E          sw          0(r29), r31
0F          seqi        r5, r1, #1    ;n=1?
10          bnez        r5, EXIT1
11          subi        r1, r1, #1    ;n--
12          jal         FACTORIAL    ;f(n-1)
.....

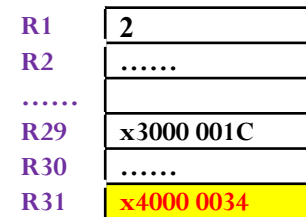
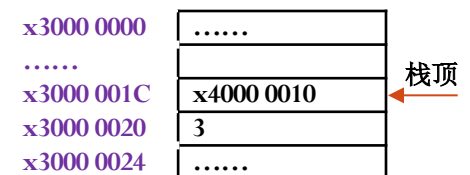
```



(3) 执行0E行后



(4) 执行12行前



(5) 执行12行后

```

0B FACTORIAL:subi      r29, r29, #4
0C          sw          0(r29), r1
0D          subi       r29, r29, #4
0E          sw          0(r29), r31
0F          seqi       r5, r1, #1    ;n=1?
10          bnez       r5, EXIT1
11          subi       r1, r1, #1    ;n--
12          jal        FACTORIAL    ;f(n-1)
.....

```

x3000 0000	
.....		
x3000 0014	x4000 0034	← 栈顶
x3000 0018	2	
x3000 001C	x4000 0010	
x3000 0020	3	
x3000 0024	

R1	2
R2
.....	
R29	x3000 0014
R30
R31	x4000 0034

(6) 执行0E行后

x3000 0000	
.....		
x3000 0014	x4000 0034	← 栈顶
x3000 0018	2	
x3000 001C	x4000 0010	
x3000 0020	3	
x3000 0024	

R1	1
R2
.....	
R29	x3000 0014
R30
R31	x4000 0034

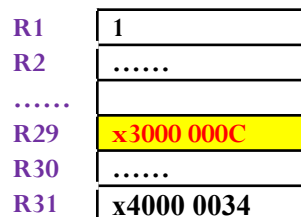
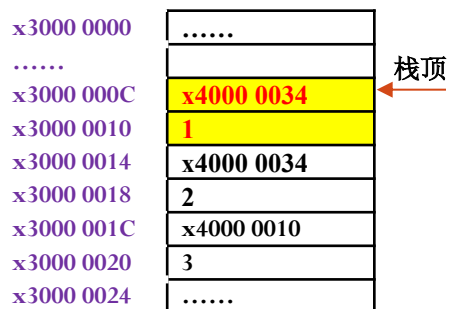
(7) 执行12行前

x3000 0000	
.....		
x3000 0014	x4000 0034	← 栈顶
x3000 0018	2	
x3000 001C	x4000 0010	
x3000 0020	3	
x3000 0024	

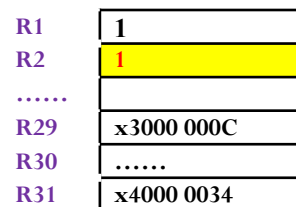
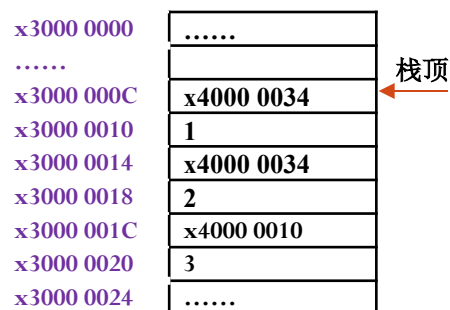
R1	1
R2
.....	
R29	x3000 001C
R30
R31	x4000 0034

(8) 执行12行后

0B FACTORIAL:	subi	r29, r29, #4
0C	sw	0(r29), r1
0D	subi	r29, r29, #4
0E	sw	0(r29), r31
0F	seqi	r5, r1, #1 ;n==1?
10	bnez	r5, EXIT1
.....		
1A EXIT1 :	addi	r2, r0, #1 ;f(1)=1
1B EXIT2 :	



(9) 执行0E行后

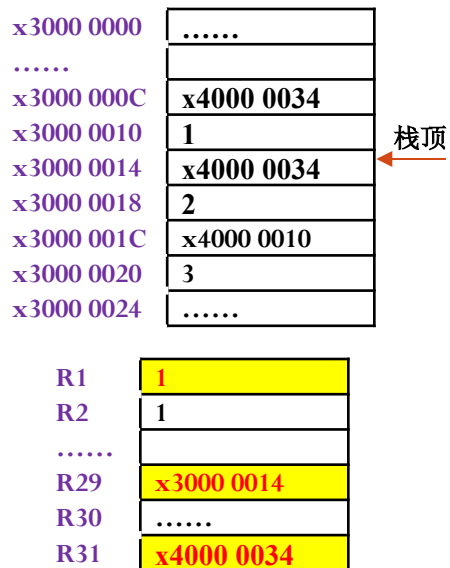


(10) 执行1B行前

```

12      jal      FACTORIAL ;f(n-1)
13      addi     r3, r2, #0 ; f(n-1)
.....
1A EXIT1 : addi     r2, r0, #1 ;f(1)=1
1B EXIT2 : lw       r31, 0(r29)
1C      addi     r29, r29, #4
1D      lw       r1, 0(r29)
1E      addi     r29, r29, #4
1F      jr       r31

```

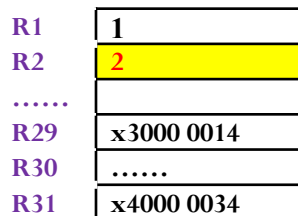
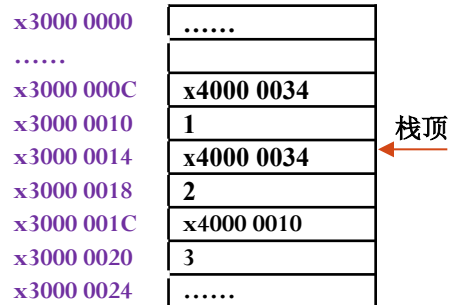


(11) 执行1E行后

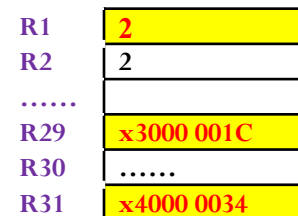
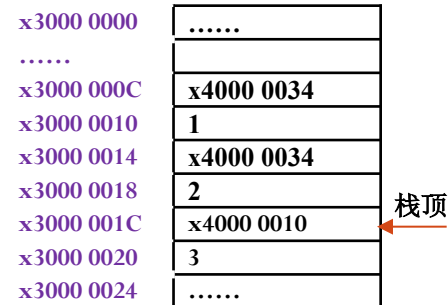

```

12      jal      FACTORIAL      ;f(n-1)
13      addi     r3, r2, #0      ;f(n-1)
14      addi     r4, r1, #1      ;n
15      andi     r2, r2, #0      ;计算n * f(n-1)
16 LOOP : beqz   r4, EXIT2
17      add      r2, r2, r3
18      subi    r4, r4, #1
19      j        LOOP
1A .....
1B EXIT2 : lw     r31, 0(r29)
1C      addi    r29, r29, #4
1D      lw      r1, 0(r29)
1E      addi    r29, r29, #4
1F      jr      r31

```



(12) 执行1B行前

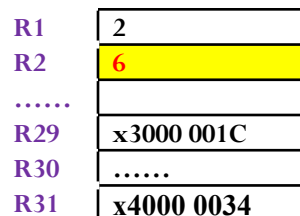
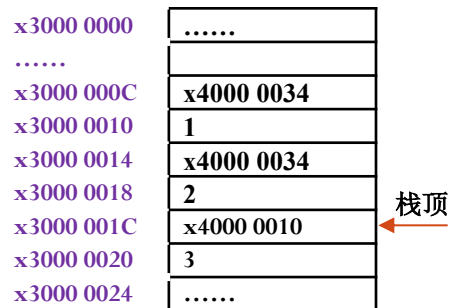


(13) 执行1E行后

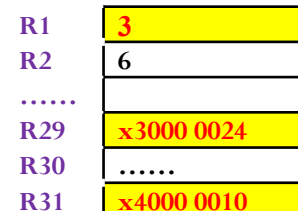
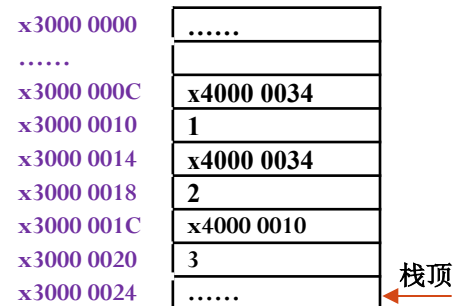
```

12      jal      FACTORIAL      ;f(n-1)
13      addi     r3, r2, #0      ; f(n-1)
14      addi     r4, r1, #1      ;n
15      andi     r2, r2, #0      ;计算n * f(n-1)
16 LOOP : beqz   r4, EXIT2
17      add      r2, r2, r3
18      subi    r4, r4, #1
19      j        LOOP
1A .....
1B EXIT2 : lw     r31, 0(r29)
1C      addi    r29, r29, #4
1D      lw      r1, 0(r29)
1E      addi    r29, r29, #4
1F      jr      r31

```



(14) 执行1B行前

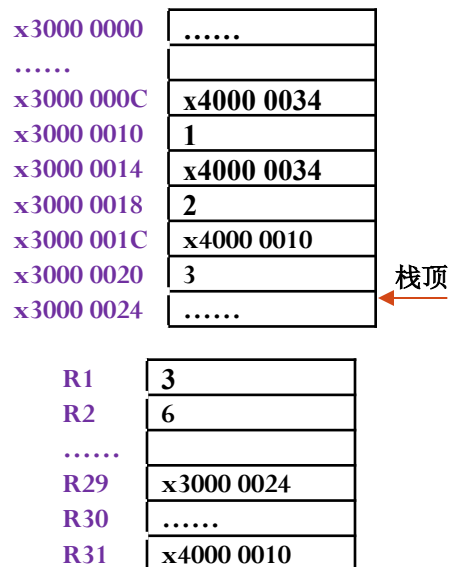


(15) 执行1E行后

```

01      .data      x30000000
02 STACK : .space   40
03;
04      .text      x40000000
05      .global    main
06 main : addi      r29, r0, STACK
07      addi      r29, r29, #40
08      addi      r1, r0, #3      ;n=3
09      jal       FACTORIAL      ;f(n)
0A      trap      x00
.....

```



(16) 执行1F行后

寄存器的保存和恢复

- R2
 - 返回值，不需要保存/压栈
- R3、R4、R5
 - 临时寄存器，对于本问题，不需要保存/压栈

C-DLX：为变量分配寄存器

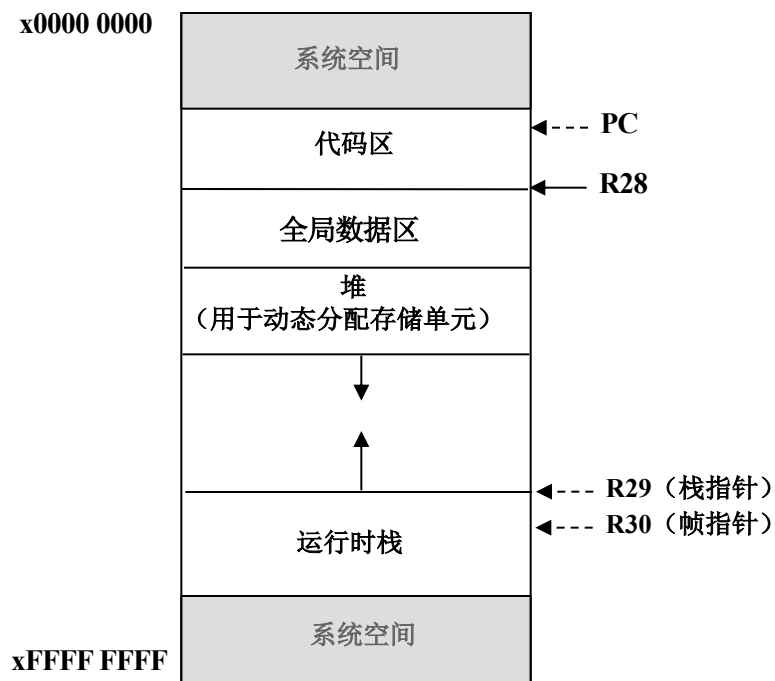
- 寄存器的访问比存储器快得多
- DLX算术/逻辑运算指令，对寄存器进行运算
- 应尽量多的使用寄存器

R0	0
R1	汇编器保留
R2、R3	返回值
R4~R7	参数
R8~R15	临时值
R16~R23	局部变量
R24、R25	临时值
R26、R27	操作系统保留
R28	全局指针
R29	栈指针
R30	帧指针
R31	返回地址

为变量分配存储空间

- 当寄存器数量不足时
 - 例如，局部变量多于8个
- 将最常用的变量保存在寄存器中，不常用的变量放到存储器中
- 基于变量的特征，为它们分配存储空间

存储器组织（类UNIX）

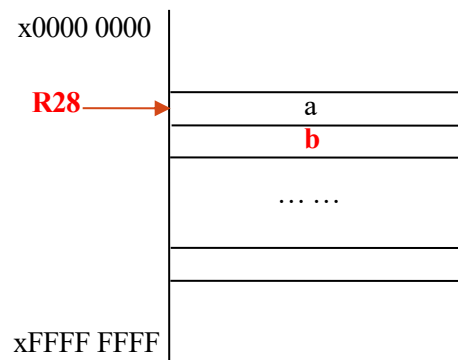


全局数据区

- 全局变量
- 静态存储类变量
 - 关键字static声明
 - 全局数据区
 - 保持对它所在的块的私有性

全局数据区示例

```
.....  
int a;  
int b;  
int main()  
{  
    .....  
    b+1;  
    .....  
}
```



- 变量距离R28的偏移量
 - a: 0
 - b: 1
- 计算b+1, 即R8←b
lw r8, 4(r28)

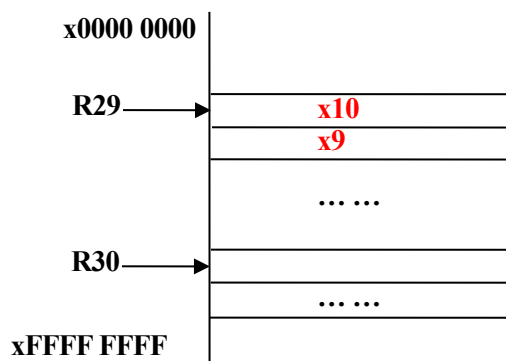
运行时栈

- 局部变量/自动存储类变量
- 函数的栈框架/活动记录
- R29，栈指针

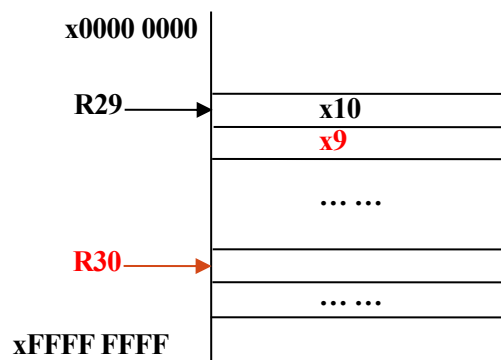
运行时栈示例

```
.....  
int main()  
{  
    int x1, x2, x3, x4, x5, x6, x7, x8, x9, x10;  
    /*对以上10个变量进行初始化*/  
    .....  
    /*输出累加和*/  
    printf("sum = %d \n", x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9  
    + x10);  
}
```

- **R16~R23** \leftarrow x1~x8



框架指针



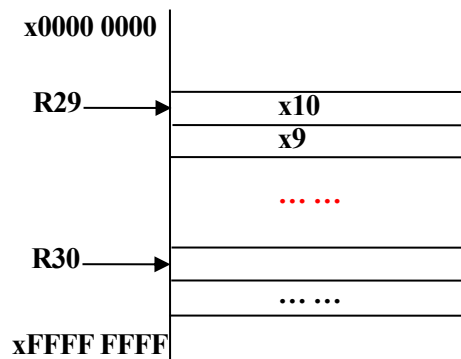
- 更方便的访问运行时栈中的变量
- 框架指针/帧指针
 - 作为基址
 - R30
- $R8 \leftarrow x9$
 - 假设x9距离R30的偏移量为-10，在R30和x9之间存储的都是整数类型的值

`lw r8, -40(r30)`

边界对齐

- 将变量分配到存储器中，还必须注意边界对齐的问题
 - char 类型

- 问题：
 - 函数的活动记录——从帧指针所指的单元到栈指针所指的单元之间，除了局部变量，还存储了哪些内容？
- 答案：与函数的编译过程有关——第15章



C语言源水平调试器

- 根据来自编译过程的信息，可以在断点处检查程序的所有执行状态，例如，变量、**存储器**、**栈**以及**寄存器**的值
- 有些调试器还允许查看编译器为源代码产生的**汇编代码**，并对汇编代码进行单步调试

IA-32

- 将变量均分配到存储器中（压栈）
 - 寄存器ebp，扩展基址指针寄存器(extended base pointer)，**帧指针**
 - 寄存器esp，**栈指针**
 - 寄存器eax，累加器

IA-32 示例

```
#include <stdio.h>
int main() {
    int x = -1;
    int y;
    y = y + x;
    printf("%d\n", y);
}
```

```
.....
_main:                                pushl    %ebp
                                      movl     %esp, %ebp
                                      andl     $-16, %esp
                                      subl     $32, %esp
                                      call     __main
                                      movl     $-1, 28(%esp)
                                      movl     28(%esp), %eax
                                      addl     %eax, 24(%esp)
                                      movl     24(%esp), %eax
                                      movl     %eax, 4(%esp)
                                      movl     $LC0, (%esp)
                                      call     _printf    leave
                                      ret
.....
```

习题

书面作业

- 11. 18 (订正: 图11. 6)