

第4章 汇编语言程序设计

4.1 单片机程序设计语言概述

4.2 汇编语言编辑和汇编及其伪指令

4.3 汇编语言程序的基本结构形式

顺序程序结构

分支程序结构

循环程序结构

4.4 汇编语言程序设计举例

汇编的含义

按照语法格式
编写源程序
*.ASM *.C

按照语法格式
将源程序翻译
成机器代码

计算机识别的
二进制代码
*.OBJ

高级语言如C++

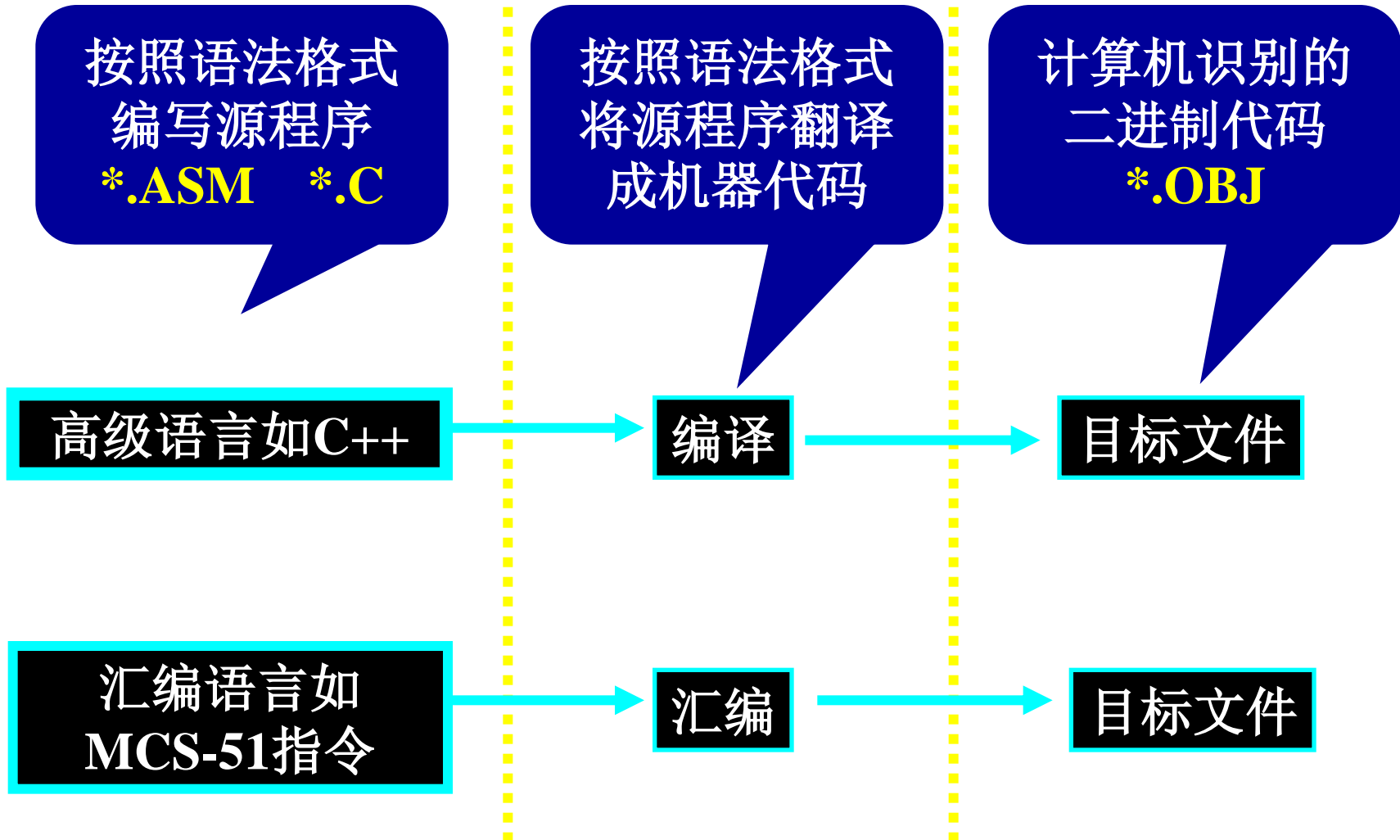
编译

目标文件

汇编语言如
MCS-51指令

汇编

目标文件



4.1 概述

一、 程序设计语言及语言处理程序

语言分： 机器语言、汇编语言和中高级语言

1. 机器语言：

硬件识别，二进制，无需翻译、直接执行，面向机器；
速度快，效率高，难以辨认和记忆，易错，难修改。

地址	机器码	源程序
		ORG 2000H
2000H	78 30	MAIN: MOV R0, #30H
2002H	E6	MOV A, @R0

...

2. 汇编语言:

由字母，数字符号组成，翻译成机器语言再由CPU执行，面向机器，编译后执行速度接近机器语言，易读，不易错，但必须熟悉指令系统，移植性差；程序精细、具体，结构紧凑，运行时间精确，高效，运算量大，实时性要求高时常用汇编。

地址	机器码	源程序
		ORG 2000H
2000H	78 30	MAIN: MOV R0, #30H
2002H	E6	MOV A, @R0

...

3. 中高级语言：

面向过程和面向对象，参照数学语言又类似日常会话语言。

高级语言中，一条高级语言指令，代替几~上百条汇编指令。直观，易学，便于移植（由编译器负责），也需经过编译、解释成机器代码后执行。C、BASIC、C++。

二、汇编语言特点及其格式

P77

标 号：操作码 操作数 ； 注释

BEGIN: MOV A, #50H ； 将立即数50H给A

- 1、**标 号**：用户定义的符号地址，便于查询和修改程序，在汇编时自动生成与该语句翻译成机器码存放在ROM单元地址相对应的16bit数。
- 2、**操作码**：规定指令所执行的操作，汇编指令中不可缺少的部分，在汇编时自动生成机器码。

标 号：操作码 操作数 ； 注释

BEGIN: **MOV** A, #50H ； 将立即数50H给A

3、操作数：是参加运算的数据或者数据的地址。

4、注 释：解释说明，增加可读性，汇编时不产生任何机器码

三 程序设计基本方法



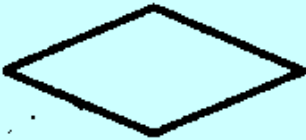


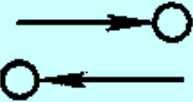
要求:

执行速度快、占用内存少、条理清晰、
阅读方便、便于移植、巧妙而实用。

步骤:

1. 分析问题
2. 确定算法
3. 设计程序流程图
4. 分配内存单元
5. 编写汇编语言源程序
6. 汇编并调试程序

流程图符号和说明:

符 号	名 称	表 示 的 功 能
	起止框	程序的开始或结束
	处理框	各种处理操作
	判断框	条件转移操作
	输入输出框	输入输出操作
	流程线	描述程序的流向
	引入引出连接线	流程的连接

4.2、MCS-51汇编语言的伪指令：

作用：告诉汇编程序如何完成汇编，不产生机器码。

1、**ORG**：起始伪指令Origin，指明程序和数据块起始地址。

指令地址	机器码	源程序
		ORG 2000H
2000H	78 30	MAIN: MOV R0, #30H
2002H	E6	MOV A, @R0
		...
		ORG 3000H
3000H	23	DB 23H, 100, 'A'
3001H	64	
3002H	41	

2、END 汇编结束伪指令

P94

例： START: ...

...

END

3、EQU —赋值伪指令。为标号或标识符赋值Equate

X1 EQU 2000H

X2 EQU 0FH

MAIN: MOV DPTR, #X1

ADD A, #X2



4、 DB 一定义字节伪指令。Define Byte 8bit

例 : DB 12H, 100, 'A'

存储: 00010010, 01100100, 01000001

5、 DW 一定义双字节伪指令。Define Word 16bit

例 : DW 2030H, 8CH , "AB"

存储: 00100000 00110000

00000000 10001100

01000001 01000010

6、DS 定义存储区伪指令 DEFINE STORAGE

从指定地址开始保留指定数目的字节单元
备用。

7、BIT 位定义伪指令

把一个可位寻址的位单元赋值给所规定的
字符名称



4.3 汇编程序的基本结构

- 顺序程序结构
- 分支程序结构
 - 单分支程序结构
 - 多分支程序结构
- 循环程序结构

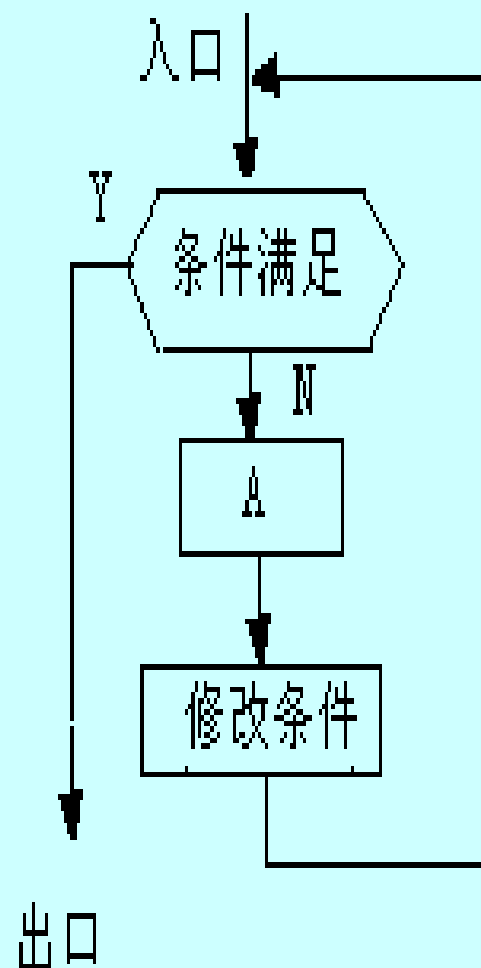
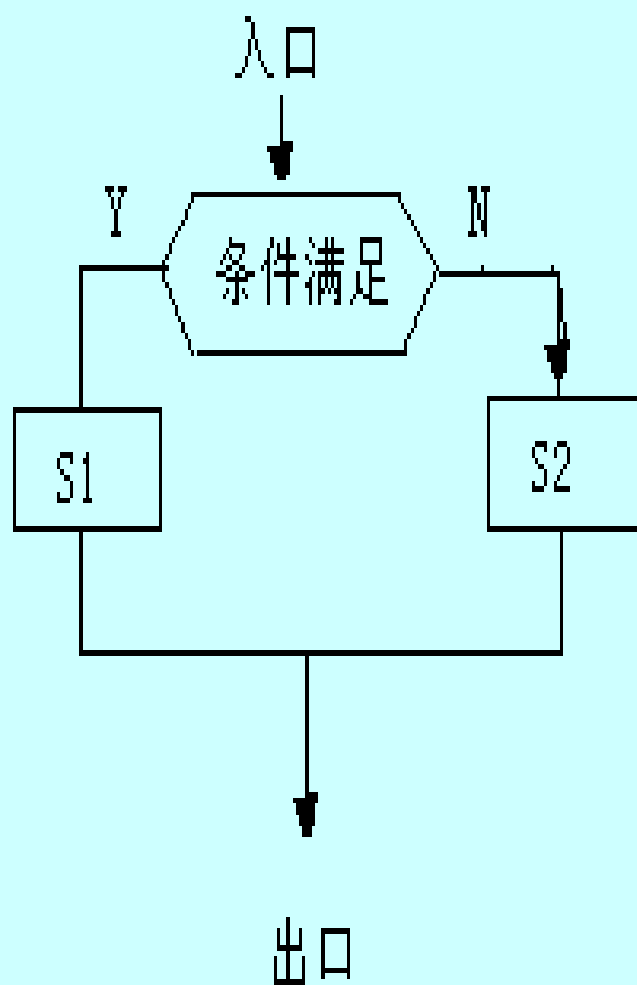
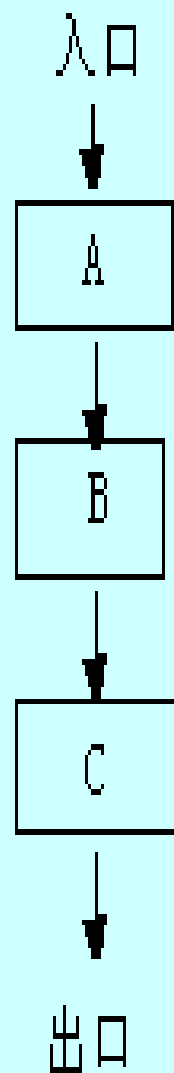


图4-3 三种基本程序结构

一、 顺序程序结构

是汇编语言程序的最简单也是最基本的程序结构。程序执行时一条接一条地按顺序执行指令，无分支、循环以及调用子程序。

```
ORG      0000H
LJMP     MAIN      ;单片机复位从该单元执行
ORG      0030H    ; 空开23页的各种入口
MAIN:    MOV      A , #30H
          ADD      A , #58H
          MOV      30H , A
          SJMP     $      ; 程序原地踏步
          END        ; 汇编结束
```

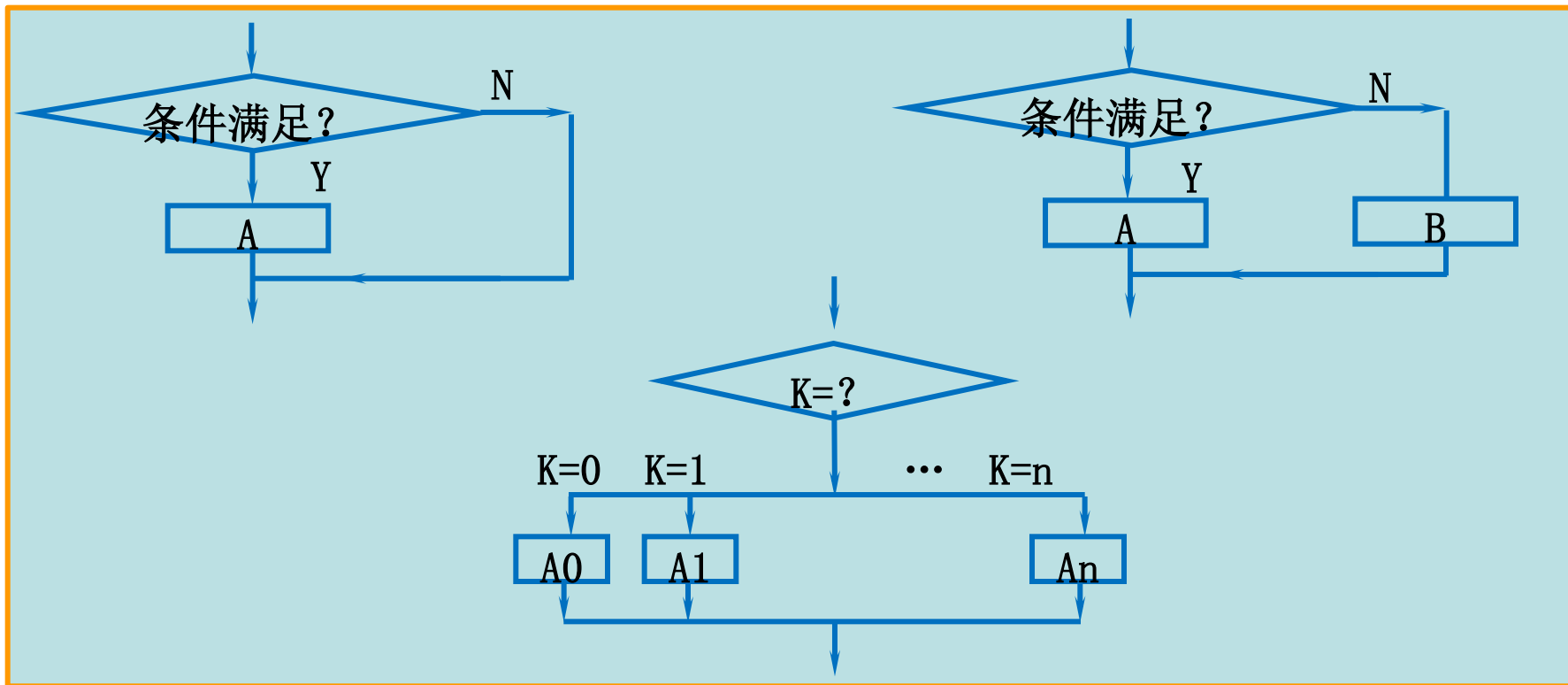

顺序程序结构分析:

找出关键的执行实际功能操作的指令，然后将其前面与该指令相关的指令归类为该指令的前期配置指令。并需要对指令占用的资源和操作的影响有全局的把握。

MOV R0, #52H	DEC R1	MOV A, @R0
MOV R1, #55H	MOV A, @R0	ADDC A, @R1
MOV A, @R0	ADDC A, @R1	MOV @R0, A
ADD A, @R1	MOV @R0, A	CLR A
MOV @R0, A	DEC R0	ADDC A, #00H
DEC R0	DEC R1	MOV 20H, A

二、 分支程序结构

- 程序分支是通过转移指令实现的，也称为选择结构。可分为单分支和多分支两类。
- 单分支程序结构，通过条件转移指令实现，即根据条件对程序的执行进行判断、满足条件则进行程序转移，不满足条件就顺序执行程序。可用指令JZ、JNZ、CJNE、DJNZ；JC、JNC、JB、JNB、JBC。



- 多分支程序是首先把分支程序按序号排列，然后按序号值进行转移。

指令: `JMP @A+DPTR`

1、单分支程序结构

在MCS-51指令系统中，通过条件判断实现单分支程序转移的指令有：JZ、JNZ、CJNE、DJNZ等。此外还有以位状态作为条件进行程序分支的指令，如JC、JNC、JB、JNB、JBC等。使用这些指令可以完成0、1、正、负，以及相等、不相等作为各种条件判断依据的程序转移。

多重单分支结构是单分支的扩展形式，通过一系列条件判断，进行逐级分支。

程序分析题:

START:	CLR	C	;为实现无借位减法
	MOV	DPTR , #ST1	;DPTR初值为ST1
	MOVX	A , @DPTR	;读ST1
	MOV	R2 , A	;R2为A复制
	INC	DPTR	;DPTR+1指向ST2
	MOVX	A , @DPTR	;读ST2
	SUBB	A , R2	; (ST2) - (ST1)
	JNC	BIG1	;判转分支
	XCH	A , R2	;有借位, A放大数ST1
BIG0:	INC	DPTR	;DPTR+1, 为ST3
	MOVX	@DPTR , A	;存A (大数) 入ST3
	RET		; 子程序返回
BIG1:	MOVX	A , @DPTR	; 无借位, A放大数ST2
	SJMP	BIG0	; 转至BIG0

例题：已知X、Y均为8位二进制有符号数，分别存在30H、31H中，试编制能实现下列符号函数的程序：

$$Y = \begin{cases} +1, & \text{当 } X > 0 \\ 0, & \text{当 } X = 0 \\ -1, & \text{当 } X < 0 \end{cases}$$

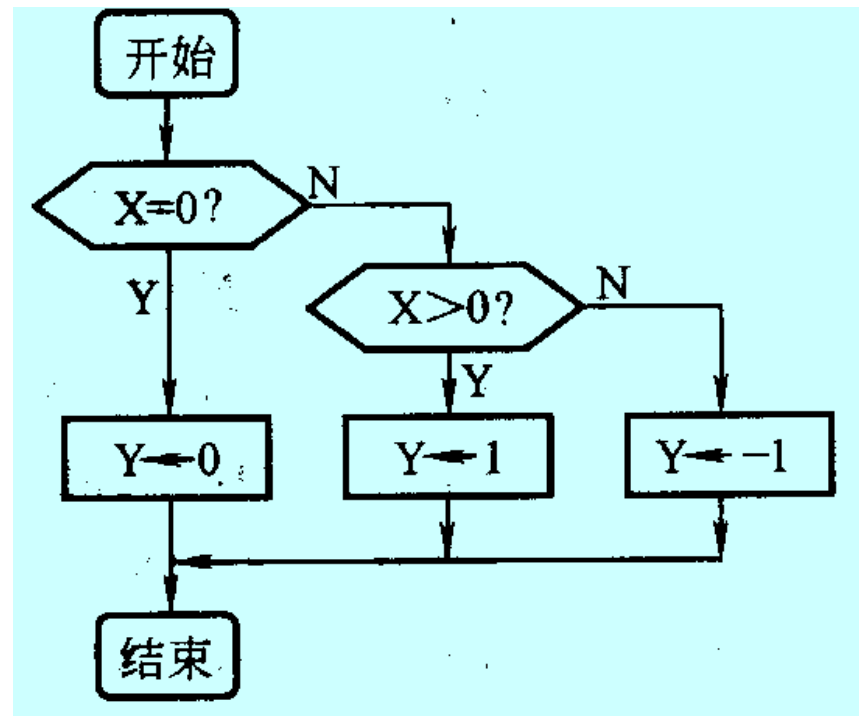
有符号数：

1000 0000b为-128

1111 1111b为-1

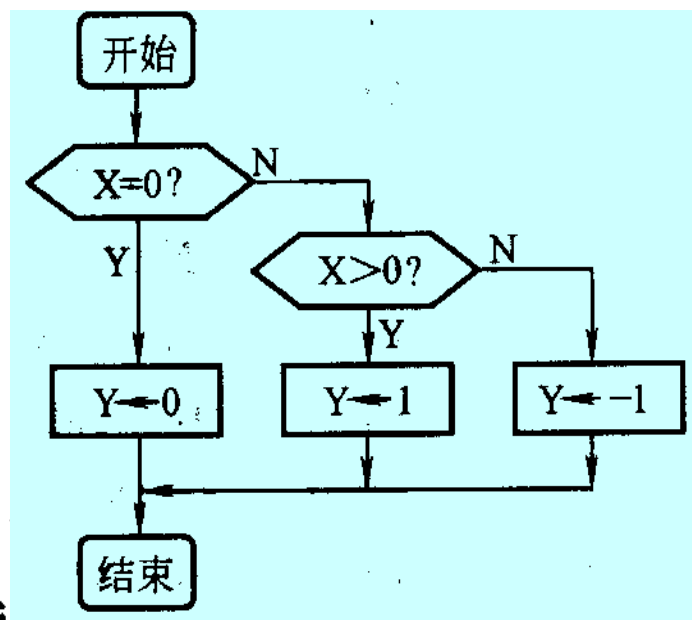
0000 0001b为 0

0111 1111b为+127



实现程序如下：

	X	EQU	30H	
	Y	EQU	31H	
SIN:	MOV	A, X		; 读 X
	JZ	SIN1		; 若 X=0, 转
	JB	Acc.7, SIN2		; 若 X<0, 转
	MOV	Y, #1		; 若 X>0, 则 1→Y
	RET			;
SIN1:	MOV	Y, #0		; C=0, 则 0→Y
	RET			;
SIN2:	MOV	Y, #0FFH		; X<0, 则 -1→Y (-1 的补码)
	RET			;



2. 多分支程序结构

常使用 `JMP @A+DPTR` 指令

1) `JMP @A+DPTR` 与数据表配合：

范围有限，256字节。

2) `JMP @A+DPTR` 与转移指令配合：

A中值为序号与转移指令字节数的乘积。

3) 利用 `RET` 指令，通过堆栈操作实现：

将分支地址从入口地址表取出后，压入堆栈，利用 `RET` 指令赋予PC分支地址。

1) JMP @A+DPTR 与数据表配合

```
MOV    A , #n
MOV    DPTR , # JMPTAB
MOVC   A , @A+DPTR
JMP    @A+DPTR
```

```
JMPTAB: DB    ROUT00—JMPTAB
          DB    ROUT01—JMPTAB
          ⋮
          DB    ROUTn —JMPTAB
```

2) JMP @A+DPTR 与转移指令配合

例： 128种分支转移程序。

功能： 根据入口条件转移到128个目的地址。

入口： (R3) =转移目的地址的序号00H~7FH。

出口： 转移到相应子程序入口。

```
JMP_128: MOV     A, R3
          RL      A
          MOV     DPTR, #JMPTAB
          JMP     @A+DPTR

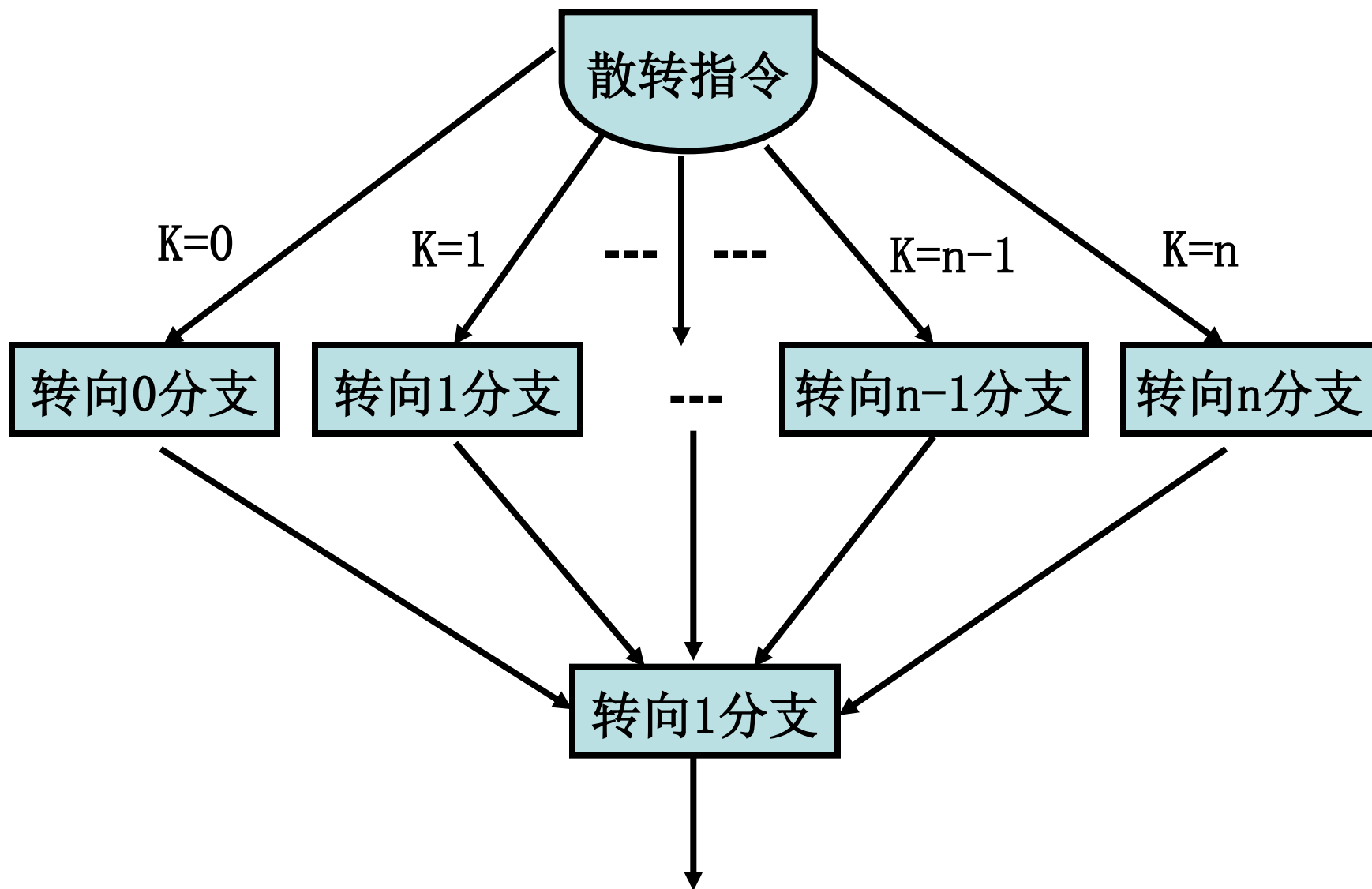
JMPTAB:  AJMP     ROUT00
          AJMP     ROUT01
          ⋮
          AJMP     ROUT7F
```

} 128个子程序首址

说明:

- 此程序要求128个转移目的地址（ROUT00 ~ ROUT7FH）必须驻留在与绝对转移指令AJMP地址相同的一个2KB存储区内。
- RL指令对变址部分乘以2，因为每条AJMP指令占两个字节。

多分支程序结构



3) 利用RET指令，通过堆栈操作实现

P82

```
MOV    DPTR , #BRTAB      BRTAB: DW ROUT00
MOV    A , R0              DW ROUT01
RL      A                  ⋮
MOV    R1,A               DW ROUT127
INC     A
MOVC   A,@A+DPTR
PUSH   ACC
MOV    A,R1
MOVC   A,@A+DPTR
PUSH   ACC
RET
```

分支程序入口地址压入堆栈，再利用返回指令将存入堆栈中的地址赋给**PC**，从而实现分支程序跳转。

作业:

97页编程题1和2

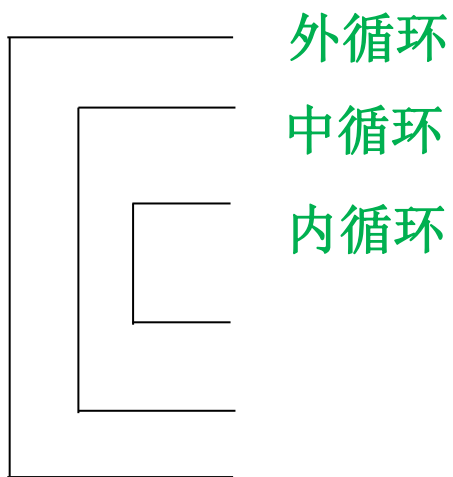
三、 循环程序

在程序运行时，有时需要连续重复执行某段程序可以使用循环程序。其结构包括四部分：

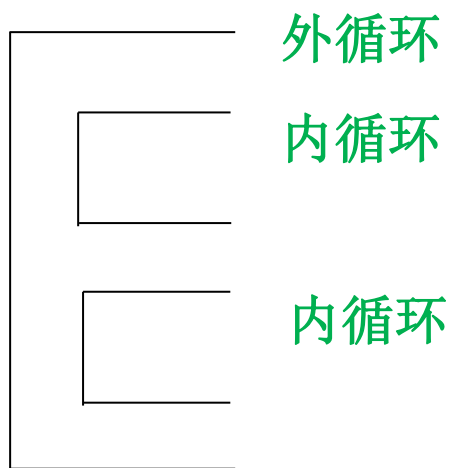
- 1、置循环初值
- 2、循环体（循环工作部分）
- 3、修改控制变量
- 4、循环控制部分

循环程序按结构形式，有单重循环与多重循环。在多重循环中，只允许外重循环嵌套内重循环。不允许循环相互交叉，也不允许从循环程序的外部跳入循环程序的内部。

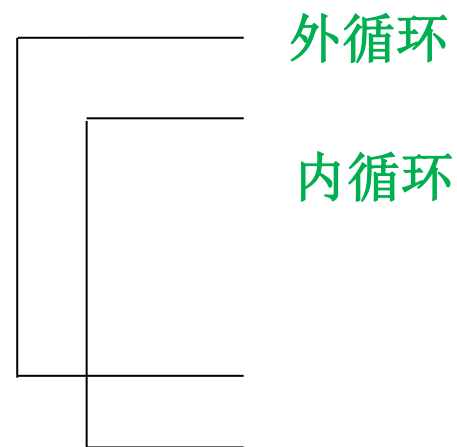
多重循环示意图



(a)
嵌套正确

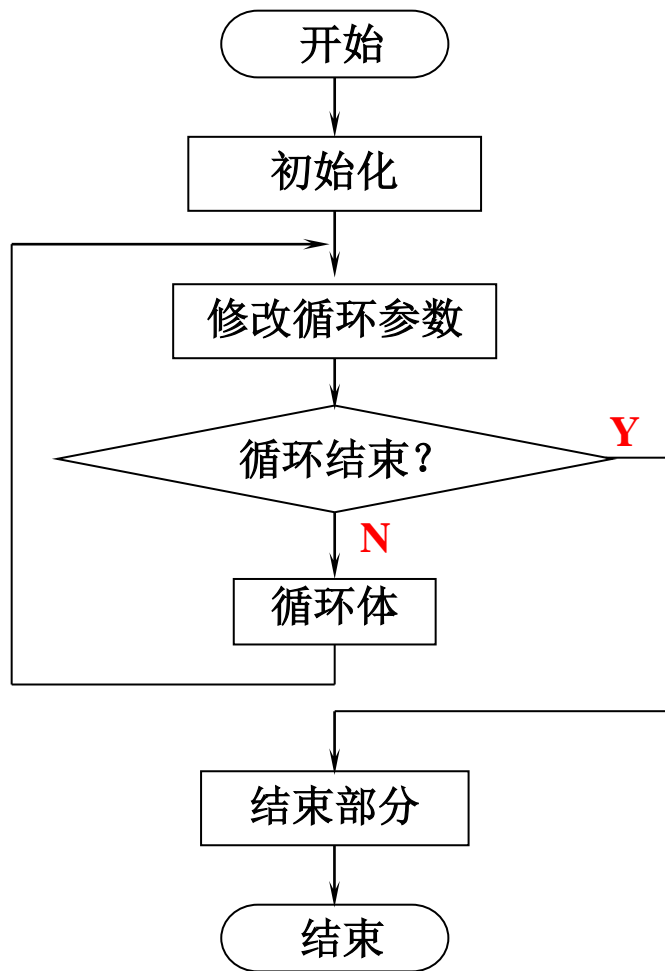


(b)
嵌套正确

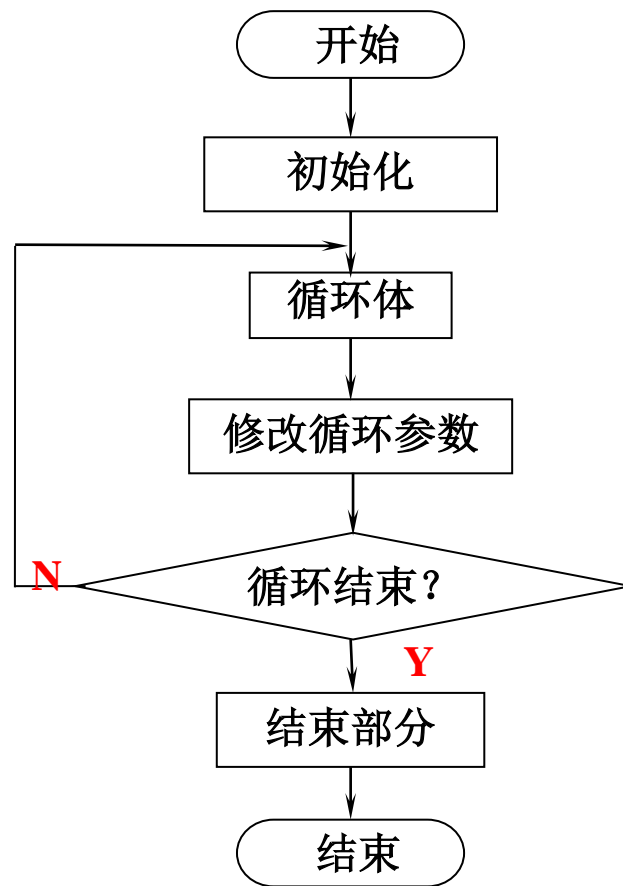


(c)
交叉不正确

循环结构程序流程图



(a) 当型循环结构



(b) 直到型循环结构

- **初始化：**建立循环初始值，如初始化地址指针，设置循环次数计数器及其他循环参数的起始值等。
- **循环体：**程序中反复要执行的部分。
- **循环修改：**对参加运算的数据或地址指针进行恰当的修改。
- **循环控制：**保证循环程序按规定的循环次数或控制循环的条件进行循环。常用条件转移语句组成，如DJNZ。
- **结束：**处理循环结果，储存结果。

MOV R0, #data ; 初值配置

MOV DPTR, #buffer

MOV R1, #20H

LOOP: MOV A, @R0

CLR C

SUBB A, #24H ; (data) - 24

JZ LOOP1 ; 条件分支, A=0跳转退出循环。

MOV A, @R0 ; A!=0, (data) 存入buffer

MOVBX @DPTR, A ; 循环体操作分析

INC DPTR ; 控制变量修改

INC R0 ; 控制变量修改

DJNZ R1, LOOP ; 循环控制语句

LOOP1: RET

4.3 应用程序设计举例

一、运算类程序

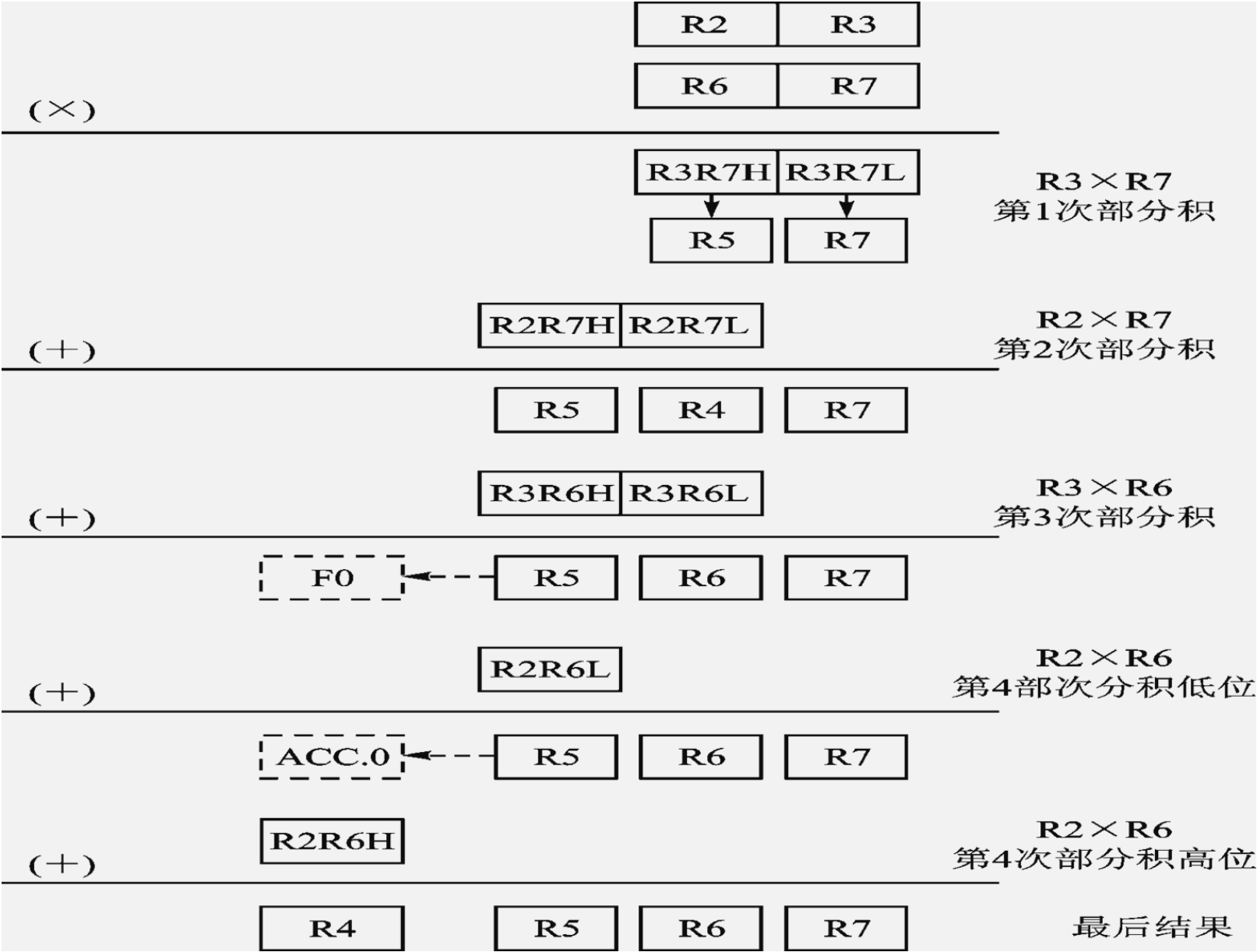
在**MCS-51** 指令系统中，给出了单字节的加、减、乘、除算术运算指令。但实际应用中还常用到双字节及多字节的算术运算程序。下面通过例子分析多字节算术运算程序的编写方法。（均是以子程序形式给出）

1、多字节无符号数加法程序

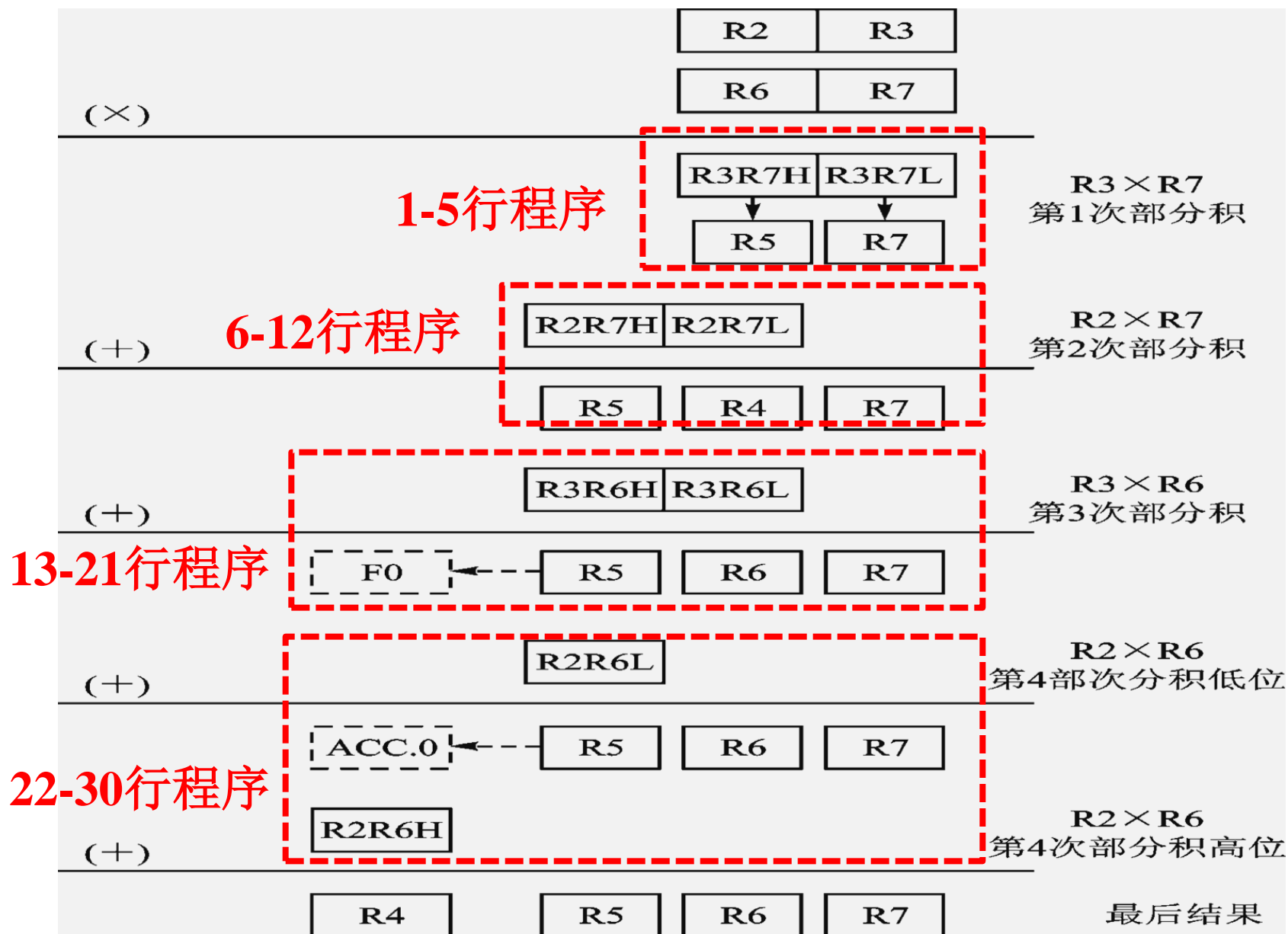
设被加数低位地址由R0指出，加数低位地址由R1指出，字节数在R2中，运算结果的和数依次存入原被加数单元，最高进位位存入用户标志F0。

解题思路：从低位开始，每个字节执行带进位相加，直到循环完R2个字节（用循环程序编写）

2、两个双字节无符号数乘法示意图



2、两个双字节无符号数乘法示意图



3、两个双字节无符号数除法示意图

被除数

— 除数**X32768**（相当于左移**15**次）

差

够减记为**1**

— 除数**X16384**（相当于左移**14**次）

差

不够减记为**0**

— 除数**X8192**（相当于左移**13**次）

差

■

○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○

■

— 除数**X2**（相当于左移**1**次）

■

差

■

— 除数**X1**（相当于左移**0**次）

■

差

■

0	0	0...	0	0	A15	A14	A13	...	A3	A2	A1	A0
0	B15	B14...	B1	B0								

X32768

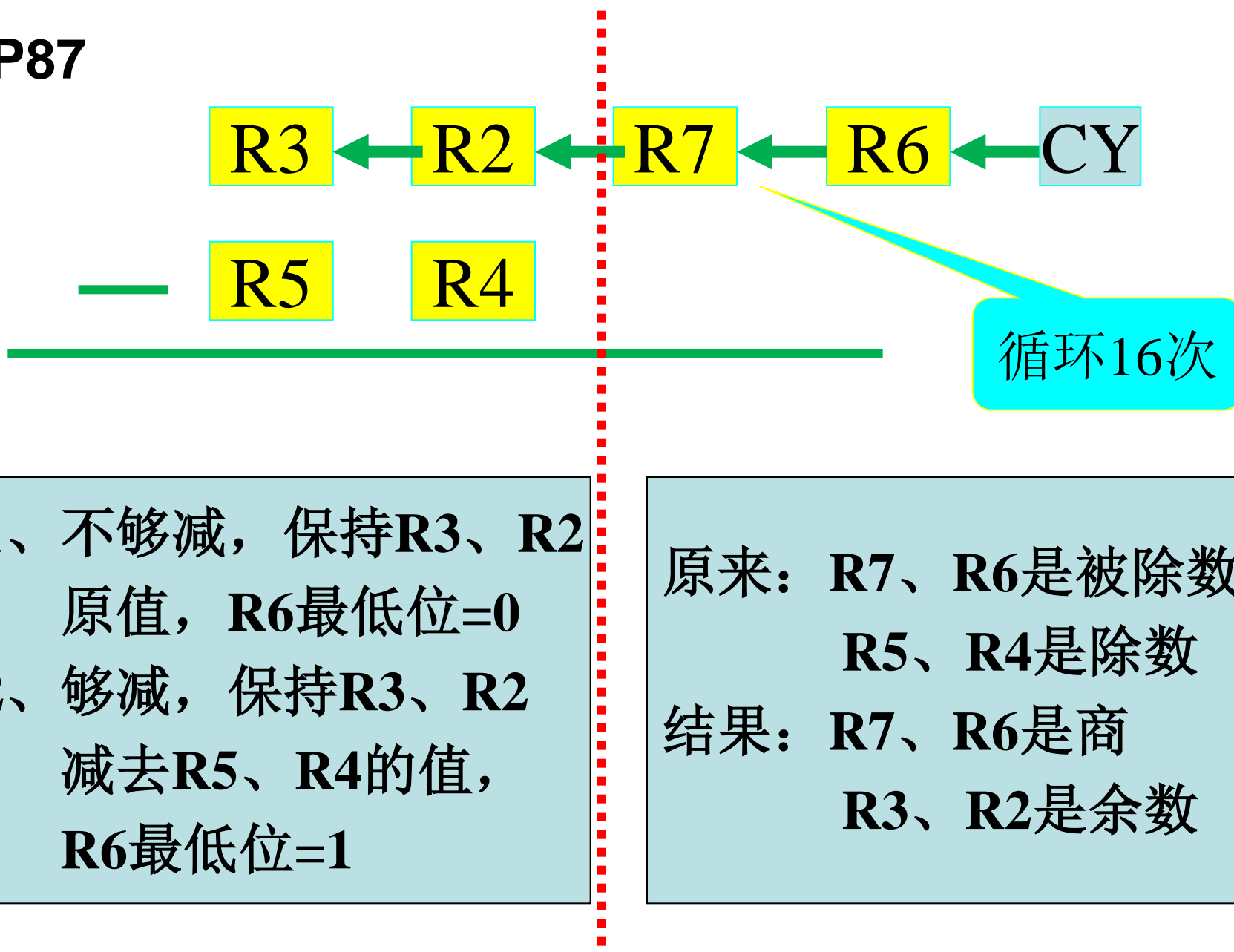
0	0	0...	0	0	A15	A14	A13	...	A3	A2	A1	A0
0	0	B15	B14...	B1	B0							

X16384

0	0	0...	0	0	A15	A14	A13	...	A3	A2	A1	A0
					B15	B14	B13	...	B3	B2	B1	B0

X1

P87



- 1、不够减，保持R3、R2原值，R6最低位=0
- 2、够减，保持R3、R2减去R5、R4的值，R6最低位=1

原来：R7、R6是被除数
R5、R4是除数
结果：R7、R6是商
R3、R2是余数

作业:

98页编程题5和7

4月14日实验内容:

上机运行作业题，即**97页1、2**和**98页5、7**四道题，自己赋不同的值，把完整程序及结果写在作业本上。

MUL AB ;低位在A，高位在B

MOV **B, A** ; A值送到B
MUL **AB** ;低位在A, 高位在B

```
SQR: MOV    B, A    ; A值送到B
      MUL    AB      ;低位在A，高位在B
      RET           ; 子程序返回
```

LCALL SQR ; 调用

SQR: MOV B, A ; A值送到B
MUL AB ;低位在A，高位在B
RET ; 子程序返回

MOV **A, DA** ; 赋值第一个数
LCALL **SQR** ; 调用

SQR: **MOV** **B, A** ; **A**值送到**B**
 MUL **AB** ;低位在**A**, 高位在**B**
 RET ; 子程序返回

MOV **A, DA** ; 赋值第一个数
LCALL **SQR** ; 调用
MOV **DC, A** ; 暂存在DC

SQR: **MOV** **B, A** ; A值送到B
 MUL **AB** ;低位在A, 高位在B
 RET ; 子程序返回

MOV **A, DA** ; 赋值第一个数
LCALL **SQR** ; 调用
MOV **DC, A** ; 暂存在**DC**
MOV **A, DB** ; 赋值第二个数
LCALL **SQR** ; 调用

SQR: **MOV** **B, A** ; **A**值送到**B**
 MUL **AB** ;低位在**A**, 高位在**B**
 RET ; 子程序返回

MOV **A, DA** ; 赋值第一个数
LCALL **SQR** ; 调用
MOV **DC, A** ; 暂存在**DC**
MOV **A, DB** ; 赋值第二个数
LCALL **SQR** ; 调用
ADD **A, DC** ; 相加
MOV **DC,A** ;再存

SQR: **MOV** **B, A** ; **A**值送到**B**
 MUL **AB** ;低位在**A**, 高位在**B**
 RET ; 子程序返回

	MOV	A, DA	;	赋值第一个数
	LCALL	SQR	;	调用
	MOV	DC, A	;	暂存在 DC
	MOV	A, DB	;	赋值第二个数
	LCALL	SQR	;	调用
	ADD	A, DC	;	相加
	SJMP	\$;	主程序结束,原地踏步,防止到下面程序
SQR:	MOV	B, A	;	A 值送到 B
	MUL	AB	;	低位在 A , 高位在 B
	RET		;	子程序返回

MOV **DA,#02H**

MOV **DB,#03H**

MOV **A, DA** ; 赋值第一个数

LCALL **SQR** ; 调用

MOV **DC, A** ; 暂存在**DC**

MOV **A, DB** ; 赋值第二个数

LCALL **SQR** ; 调用

ADD **A, DC** ; 相加

SJMP **\$** ;主程序结束,原地踏步,防止到下面程序

SQR: **MOV** **B, A** ; **A**值送到**B**

MUL **AB** ;低位在**A**, 高位在**B**

RET ; 子程序返回

```

    ORG      0000H
    LJMP     MAIN
    ORG      0030H
MAIN: MOV     DA,#02H
      MOV     DB,#03H
      MOV     A, DA    ; 赋值第一个数
      LCALL   SQR      ; 调用
      MOV     DC, A    ; 暂存在DC
      MOV     A, DB    ; 赋值第二个数
      LCALL   SQR      ; 调用
      ADD     A, DC     ; 相加
      SJMP    $         ;主程序结束,原地踏步,防止到下面程序
SQR:  MOV     B, A      ; A值送到B
      MUL     AB        ;低位在A, 高位在B
      RET          ; 子程序返回
    END

```

DA EQU 20H
DB EQU 21H
DC EQU 22H

ORG 0000H

LJMP MAIN

ORG 0030H

MAIN: MOV DA,#02H

MOV DB,#03H

MOV A, DA ; 赋值第一个数

LCALL SQR ; 调用

MOV DC, A ; 暂存在DC

MOV A, DB ; 赋值第二个数

LCALL SQR ; 调用

ADD A, DC ; 相加

SJMP \$;主程序结束,原地踏步,防止到下面程序

SQR: MOV B, A ; A值送到B

MUL AB ;低位在A, 高位在B

RET ; 子程序返回

END