第五章　顺序、分支、循环程序设计

学习目的

- 掌握转移指令与循环指令的使用方法，能够在程序中使用各种程序结构。
- 掌握汇编语言程序设计的完整步骤，能够上机调试程序。

5.1 顺序结构程序设计

- 顺序结构：指令按照内存中的地址顺序依次被执行，没有任何流程改变。

- 例1 计算 Z =（3X+Y-5）/ 2 （其中的变量为无符号数）

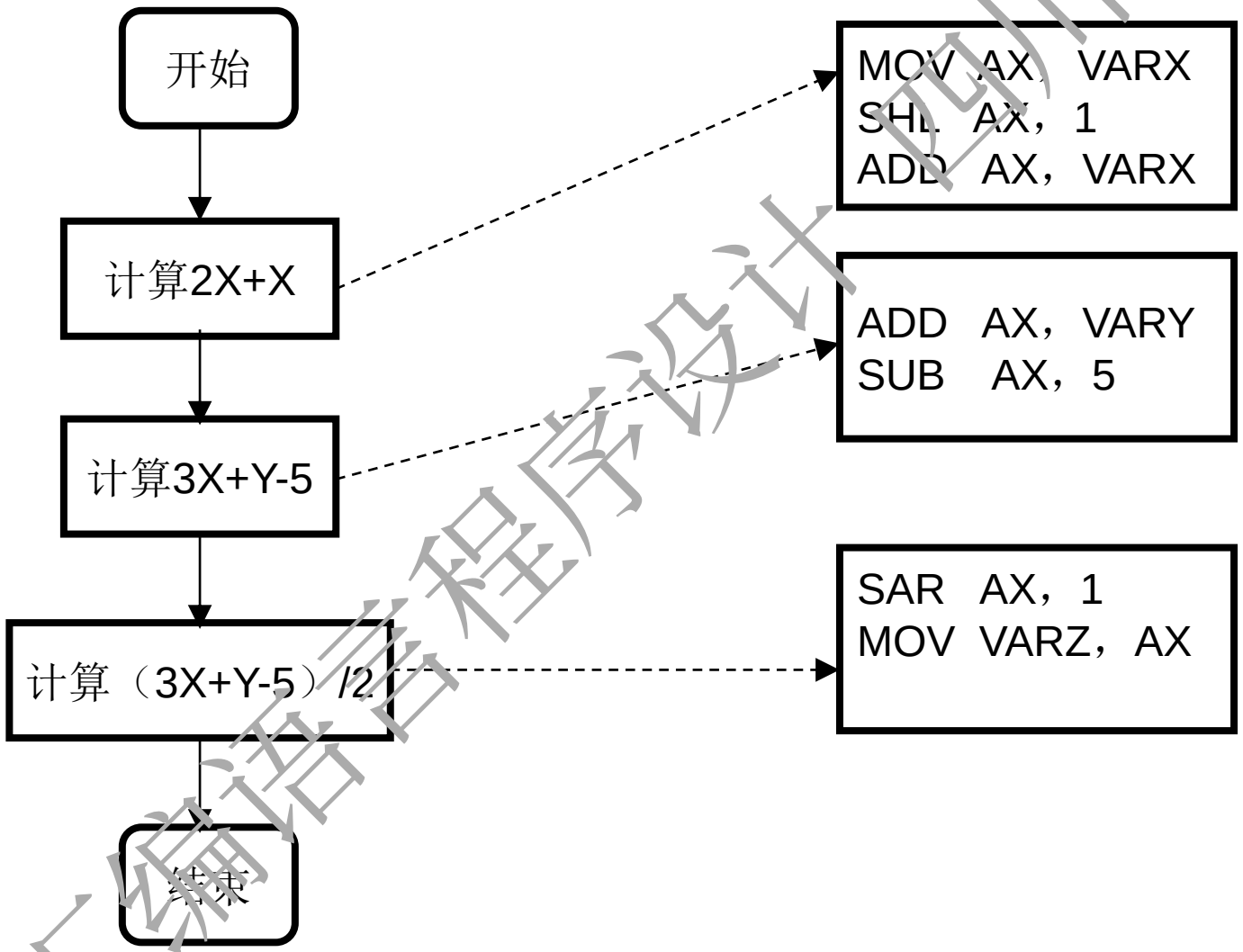
（1）数据分析

- 表达式中有三个变量X、Y、Z，可以使用三个内存单元来表示。
- 未指明变量的变化范围，使用字节、字为单位均可，这里以字为单位。

（2）算法分析

- 3X ： 2X+X　　一条移位指令和一条加法指令
- 3X+Y-5 ：　一条加法指令和一条减法指令
- （3X+Y-5）/ 2 ：可使用右移指令
- 程序应为顺序结构，按照表达式规定的顺序使用加法、减法、移位指令可实现全部运算。

（3）绘制流程图并写出程序主体



编制完整程序的余下步骤

- （4）选择一种程序框架，把程序主体添加到框架中。
- （5）汇编过程：使用MASM.EXE对源程序文件（ASM）进行汇编，检查并修改语法错误，生成目标代码文件（OBJ）。
- （6）连接过程：使用LINK.EXE连接工具把OBJ文件组织为可执行文件（EXE文件）。
- （7）使用DEBUG调试可执行程序，观察并修改程序中的逻辑错误。

5.2 分支程序设计

- 若程序需对计算结果或测试条件进行判断，根据判断结果来决定程序流程，则采用分支结构。

- 汇编语言中，实现分支结构的指令是转移类指令，这类指令通过修改CS和IP的内容来改变程序执行的流程。

- 循环结构可看作分支结构的一种特例。

5.2.1 转移指令

- 转移指令分为无条件和条件两类。
- 无条件转移指令一定改变程序流程，改变下一条将要执行的指令地址。
- 条件转移指令会判断特定的标志位或寄存器，如果满足条件，则改变程序流程，如果不满足则不会改变程序流程。
- 两种转移指令的功能都不可忽视，在程序中都是经常使用的。

（1）无条件转移指令

- 指令格式：JMP　目标地址

- 功能：无条件的修改IP或者CS、IP中的内容，改变下一条将要执行的指令地址。即跳转到目标地址继续执行程序。

- 标志位影响：无

（1）无条件转移指令

```
        JMP    L1
        MOV    AX，0
        …
L1:     MOV    AX，0FFFFH
        …
```

- 执行JMP指令后，程序流程改变到L1标号指明的地址。

（1）无条件转移指令

- JMP指令的转移类型分为两种：段内转移和段间转移。

- （1）段内转移

- JMP指令本身和目标地址在同一代码段内，只需修改IP的内容，而不需要改变CS的内容。

（1）无条件转移指令

- 1）段内直接转移
- 格式：JMP 标号
- 机器指令由两部分构成，OPCODE和DISP，OPCODE是操作码，DISP是一个8位或16位的补码。

1）段内直接转移指令

- DISP表示的含义是标号地址相对于JMP后一条指令的字节距离。
- DISP = 标号偏移量 - JMP后一条指令偏移量
- 上面的计算是汇编程序在汇编过程中完成的。

1）段内直接转移指令

- 标号地址 > JMP后一条指令地址：
- DISP > 0

- 标号地址 < JMP后一条指令地址：
- DISP < 0

- 功能：IP <-（IP）+ DISP

1）段内直接转移指令

- （IP）+DISP

- 该加法是两个补码的加法。DISP可以是正数或负数，但是（IP）总是理解为正数。

- 把第16位（超出字的范围）理解为符号位，（IP）的第16位总是理解为0，DISP的第16位由它自己的符号位决定。

1）段内直接转移指令

- 例：（IP）=1100101001101011
- DISP = 11100110

- （IP）+DISP：

- 0 11001010 01101011
- \+ 1 11111111 11100110
- 0 11001010 01010001

- 注意，相加的结果总是一个正数，补码加法完毕以后应当重新把（IP）看作无符号数。

短转移与长转移

- DISP字段是由汇编程序在汇编过程中计算并生成的。
- 按照DISP的长度，段内直接转移指令又可分为短转移和长转移两种类型。
- 若DISP在-128—127范围内，则只占用一个字节。这种情况称为短转移。
- 若DISP超过了-128--127范围，则DISP占用一个字。这种情况称为长转移，其转移范围为-32768——32767。

2）段内间接转移

- JMP指令针对的目标偏移量存放在16位通用寄存器或字内存单元中。

- 指令格式：JMP　通用寄存器名称
- 　　JMP　字内存单元

- 功能：IP <=（通用寄存器）
- 　　IP <=（EA）

2）段内间接转移

- 例：

- JMP　BX
- 功能：IP <=（BX）

- JMP　WORD PTR [SI]
- 功能：IP <=（DS：SI）

段内直接与段内间接的区别

- 注意段内直接转移和段内间接转移的区别。
- 段内直接转移：由机器指令的DISP字段给出目标地址相对于JMP后一条指令的带符号字节距离，功能为IP <=（IP）+ DISP。
- 段内间接转移：由16位通用寄存器或字内存单元直接给出IP寄存器新的内容。

（2）段间转移

- 标号地址和JMP指令不在同一代码段内，这类转移必须同时修改CS和IP的内容，才能完成正确的转移操作。

1）段间直接转移

- 指令格式：JMP　FAR类型的标号
- JMP　FAR　PTR　标号

- 第二种格式是强制指定标号的临时类型属性为FAR，无论该标号定义为什么类型，都作为FAR类型来引用。

1）段间直接转移

- 机器指令中除OPCODE字段外，还有两个16位的字段，分别表示目标地址的偏移量和段基值。

- 功能：IP <= 目标地址偏移量
- CS <= 目标地址段基值

- 同时修改CS、IP中的内容，实现段间转移。

2）段间间接转移

- 该转移方式所用的目标地址只能是存放在双字内存单元中的内容。
- 指令格式：
- JMP　DWORD PTR 双字单元起始偏移量
- 功能：IP <=（EA）
- 　　CS <=（EA+2）

2）段间间接转移

- 例：
- …
- ADR1　DD　L1
- …
- JMP DWORD PTR ADR1

- 功能：实现段间转移，目标地址为L1标号所在位置，逻辑地址存放在ADR1双字单元中。

2）段间间接转移

- 例：JMP DWORD PTR [BX][DI]

- 双字单元有效地址：EA=（BX）+（DI）
- 采用基址变址寻址方式。

- 功能：IP <=（EA）
- CS <=（EA+2）

（2）条件转移指令

- 功能：
- 根据前一条指令所影响的某些标志位或寄存器状态作为判断条件；

- 当条件满足时，把程序流程转移到指定的目标地址；

- 当条件不满足时，不改变程序流程，CPU会顺序执行下一条指令。

（2）条件转移指令

- 统一的汇编指令格式：JXX　目标地址

- 标志位影响：无（只使用标志位作为判断条件，不会去修改它）

（2）条件转移指令

- 机器指令格式：总是占用2个字节，OPCODE与带符号的相对位移量（DISP）。

- 由于DISP仅有8位，转移范围一定是 -128 — 127字节，只能是短转移，这一特征对于所有条件转移指令是一致的。

- 如果需要条件转移的转移范围得到扩大，那么必须结合使用无条件转移。

（2）条件转移指令

- 在执行条件转移指令前，一般需要执行一条算术运算指令或逻辑运算指令对特定标志位产生有意义的影响。

- 根据所判断的条件，条件转移指令可分为三类：单条件转移、无符号数条件转移，带符号数条件转移。

（1）单条件转移指令

- 这一类条件转移指令只对某一个标志位或寄存器的状态进行判断，并决定是否进行转移。

1）单标志位转移

- 单标志位转移指令总共有5对，能对5个标志位：CF、ZF、SF、OF、PF各自的状态进行判断。

- 例：
- JC　　L1
- JNC　L1

1）单标志位转移

- 例1 编写一段程序判断两个字节变量DB1和DB2的内容是否相同，如果相同则把AL清0，否则置全1。

```
        MOV  AL，DB1
        CMP  AL，DB2
        JZ    L1
        MOV  AL，0FFH
        JMP   L2
        ；无条件转移对于实现分支结构也是必不可少的
L1：MOV  AL，00H
L2：MOV  AH，4CH
        INT  21H
```

1）单标志位转移

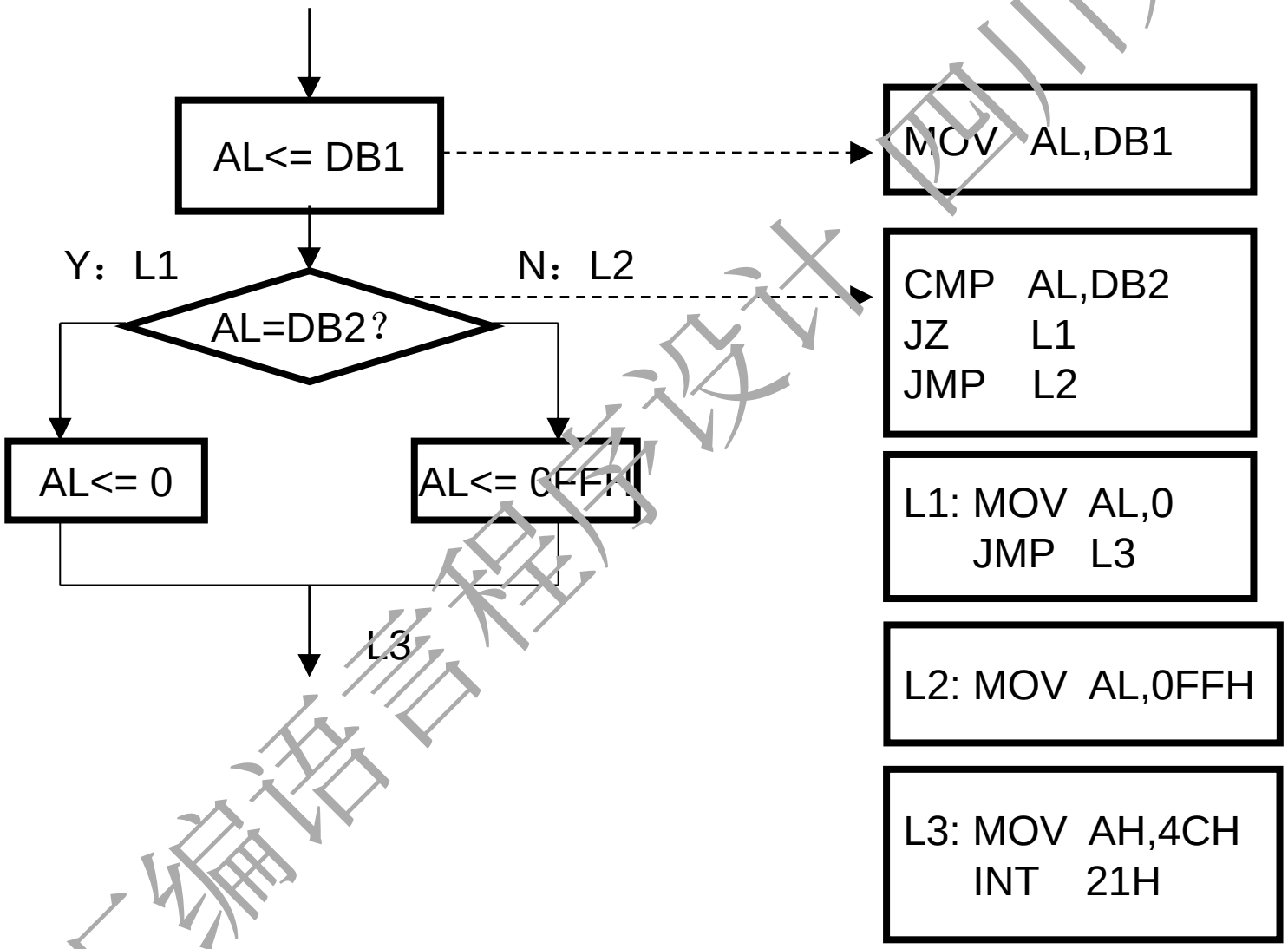
- 注意分析，上面的条件转移有什么样的缺陷。

- 如果JZ L1指令和L1标号之间的字节距离超过127，那么在汇编过程中就会出现语法错误。

1）单标志位转移

- 条件转移指令不适合直接用于距离较远的转移。
- 一般来说，应该结合无条件转移指令在分支结构中实现长距离转移。
- 就上面的例子，可通过流程图来得到合理的源程序。

根据流程图写出相关程序主体



2）另一种单条件转移

- 指令格式：JCXZ　目标地址
- 功能：判断（CX）是否为0，若是则跳转到目标地址，否则顺序执行下一条指令。
- 该指令主要用于和循环指令配合使用，以实现程序中循环结构。
- 因为循环指令使用CX寄存器作为计数器实现计数循环。

JCXZ

- JCXZ是对（CX）进行判断，而不是ZF标志位。例：
- MOV AX，34H
- MOV CX，34H
- CMP CX，AX
- JCXZ L1
- …
- L1:
- …

- 程序中，JCXZ指令并不会跳转到L1，因为（CX）不等于0，ZF标志不会影响它的判断。

（2）无符号数条件转移指令

- 一组用于判断两个无符号数大小关系，从而决定是否执行转移的指令。
- 使用前提：
- 使用这组指令前，要保证上一条影响标志位的指令是CMP指令；
- 并且程序员把两个参与比较的操作数看作无符号数。

（2）无符号数条件转移指令

- 这组指令在逻辑上把CMP指令的目的操作数称为A数据，把源操作数称为B数据。

- 通过CF标志和ZF标志的状态来判断A、B两个数据的大小关系，决定是否跳转。

（2）无符号数条件转移指令

- JA：A > B 成立则转移
- JNBE：A ≤ B 不成立则转移
- 这两个助记符含义相同，对应同一条机器指令。

- 判断规则：CF=0 AND ZF=0，转移，否则不转移。

- CF=0：A-B最高位不产生借位，够减，A≥B
- ZF=0：A-B结果不为0，A≠B
- 综合分析所得：A>B

（2）无符号数条件转移指令

- JAE：A≥B成立则转移
- JNB：A<B不成立则转移
- 两个助记符对应同一条机器指令。

- 判断规则：CF=0 OR ZF=1，转移，否则不转移。

- CF=0：A-B最高位不产生借位，够减，A≥B
- ZF=1：A-B结果为0，A=B
- 综合分析所得：A≥B

（2）无符号数条件转移指令

- JB：A<B成立则转移
- JNAE：A≥B不成立则转移
- 两个助记符对应同一条机器指令。

- 判断规则：CF=1 AND ZF=0，转移，否则不转移。

- CF=1：A-B最高位产生借位，不够减，A<B
- ZF=0：A-B的结果不为0，A≠B
- 综合分析所得：A<B

（2）无符号数条件转移指令

- JBE：A≤B成立则转移
- JNA：A>B不成立则转移
- 两个助记符对应同一条机器指令。

- 判断规则：CF=1 OR ZF=1，转移，否则不转移。

- CF=1：A-B最高位产生借位，不够减，A<B
- ZF=1：A-B的结果为0，A=B
- 综合分析所得：A≤B

（2）无符号数条件转移指令

- 例2　数据段中的ARY数组中存放了10个字节型的无符号数，从数组中找出最大的数保存到MAX字节单元中。

例2 （1）数据分析

- 数组ARY需要10个字节的空间。

- MAX字节单元用于存放最终结果。

例2（2）算法分析

- 1）把数组中的第0个数据首先存放到一个寄存器中（这里选用AL）。

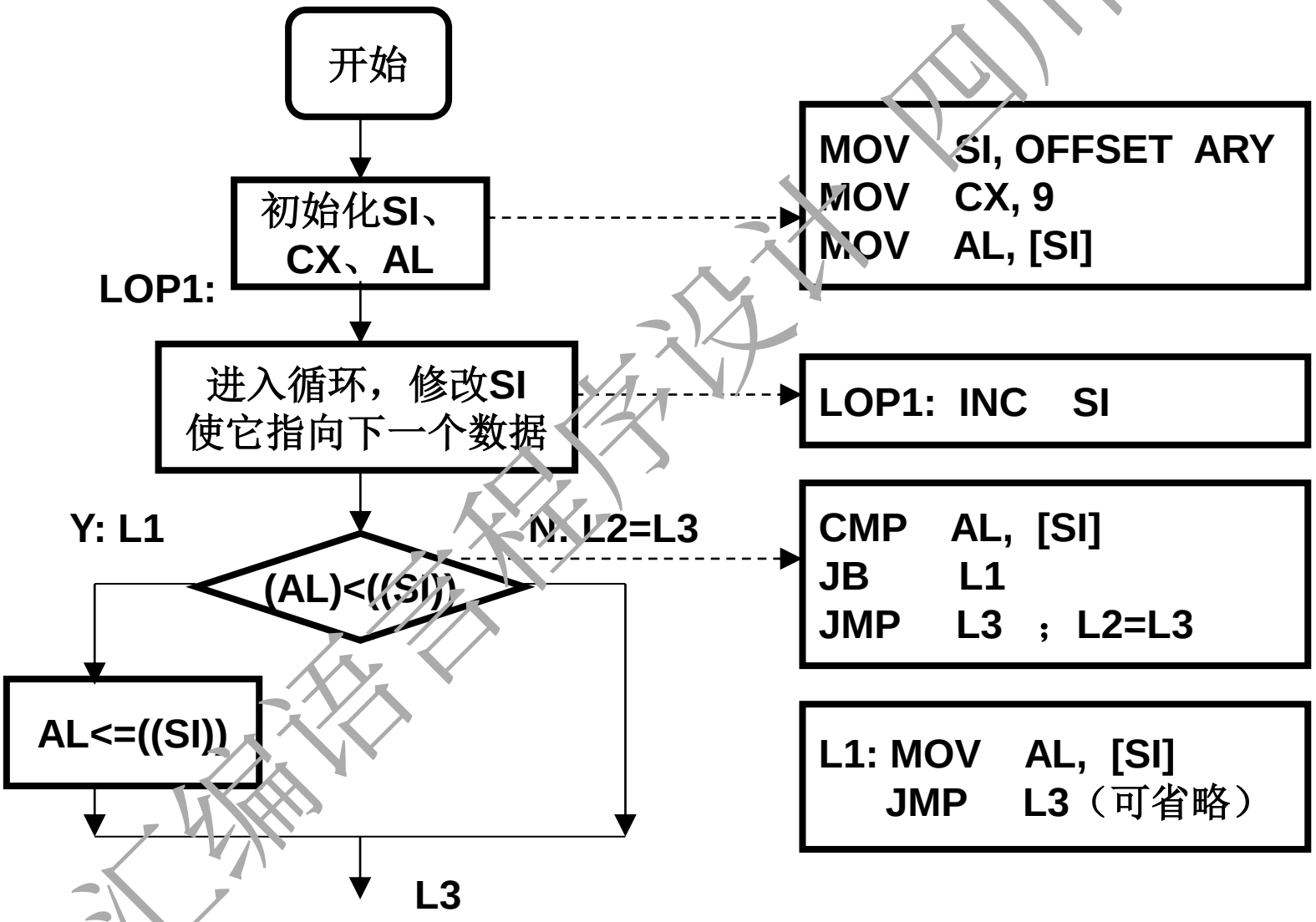
- 2）将寄存器中数据与数组中每一个数据按顺序作比较，每遇到一个大于寄存器中数据的数组元素，则用它替换寄存器中数据，保持寄存器中数据总是当前最大值。

例2（2）算法分析

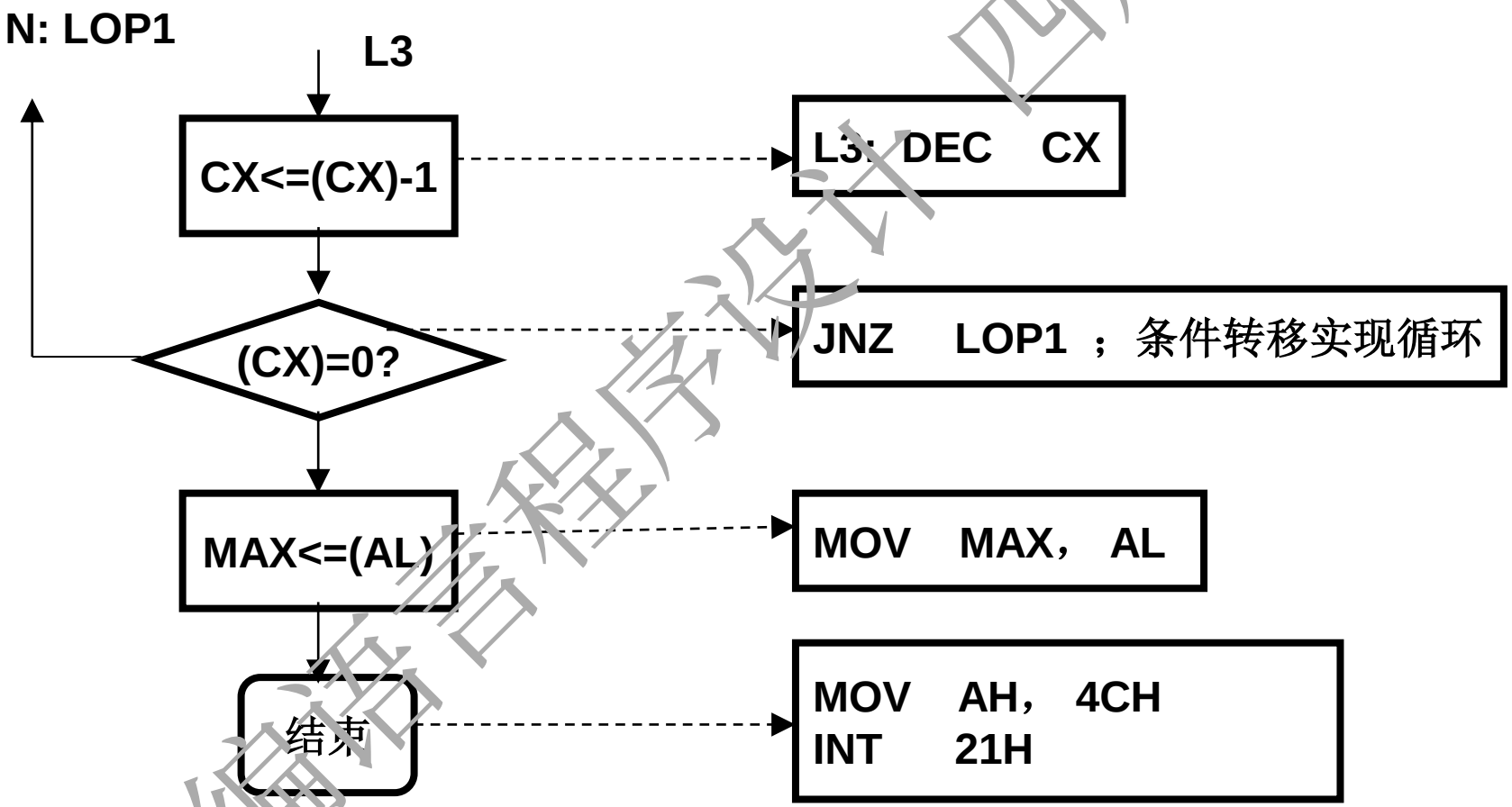
- 3）由于需要逐个访问数组中的元素，需要使用一个基址或变址寄存器作为一维数组的浮动下标，这里选用SI；

- 4）需要一个计数器，对访问次数进行记数，保证对数组的访问不会越界，这里数组中有10个元素，使用CX来完成记数的功能。

（3）绘制流程图并写出程序主体



（3）绘制流程图并写出程序主体



（3）带符号数条件转移指令

- 一组用于判断两个带符号数大小关系，从而决定是否执行程序流程转移的指令。

- 使用前提：
- 使用这组指令之前，要保证上一条影响标志位的指令是CMP指令，

- 并且程序员把两个参与比较的操作数看作带符号数。

（3）带符号数条件转移指令

- 这组指令在逻辑上把CMP指令的目的操作数称为A数据，把源操作数称为B数据。

- 通过OF标志、SF标志和ZF标志的状态来判断A、B两个数据的大小关系，决定是否跳转。

（3）带符号数条件转移指令

- JG：A>B成立则转移
- JNLE：A≤B不成立则转移
- 判断规则：SF=OF AND ZF=0，转移，否则不转移。

- SF=OF：（参见教案对CMP指令的分析）A-B的正确结果应该是一个正数或者是0，A≥B

- ZF=0：A-B结果不为0，A≠B
- 综合分析所得：A>B

（3）带符号数条件转移指令

- JGE：A≥B成立则转移
- JNL：A<B不成立则转移
- 判断规则：SF=OF OR ZF=1，转移，否则不转移。

- SF=OF： A≥B
- ZF=1：A-B结果为0，A=B
- 综合分析所得：A≥B

- 逻辑上ZF=1这个条件可以去掉。

（3）带符号数条件转移指令

- JL：A<B成立则转移
- JNGE：A≥B不成立则转移
- 判断规则：SF≠OF AND ZF=0，转移，否则不转移。

- SF≠OF：A-B的正确结果应该是负数，A<B
- ZF=0：A-B的结果不为0，A≠B
- 综合分析所得：A<B

- 逻辑上ZF=0这个条件可以去掉。

（3）带符号数条件转移指令

- JLE：A≤B成立则转移
- JNG：A>B不成立则转移

- 判断规则： SF≠OF OR ZF=1，则转移，否则不转移。

- SF≠OF：A<B
- ZF=1：A=B

- 综合结果：A≤B

（3）带符号数条件转移指令

- 例3. 修改例2，把数组中存放的无符号数改成带符号数，要求不但找出数组中最大值，而且找出最小值。

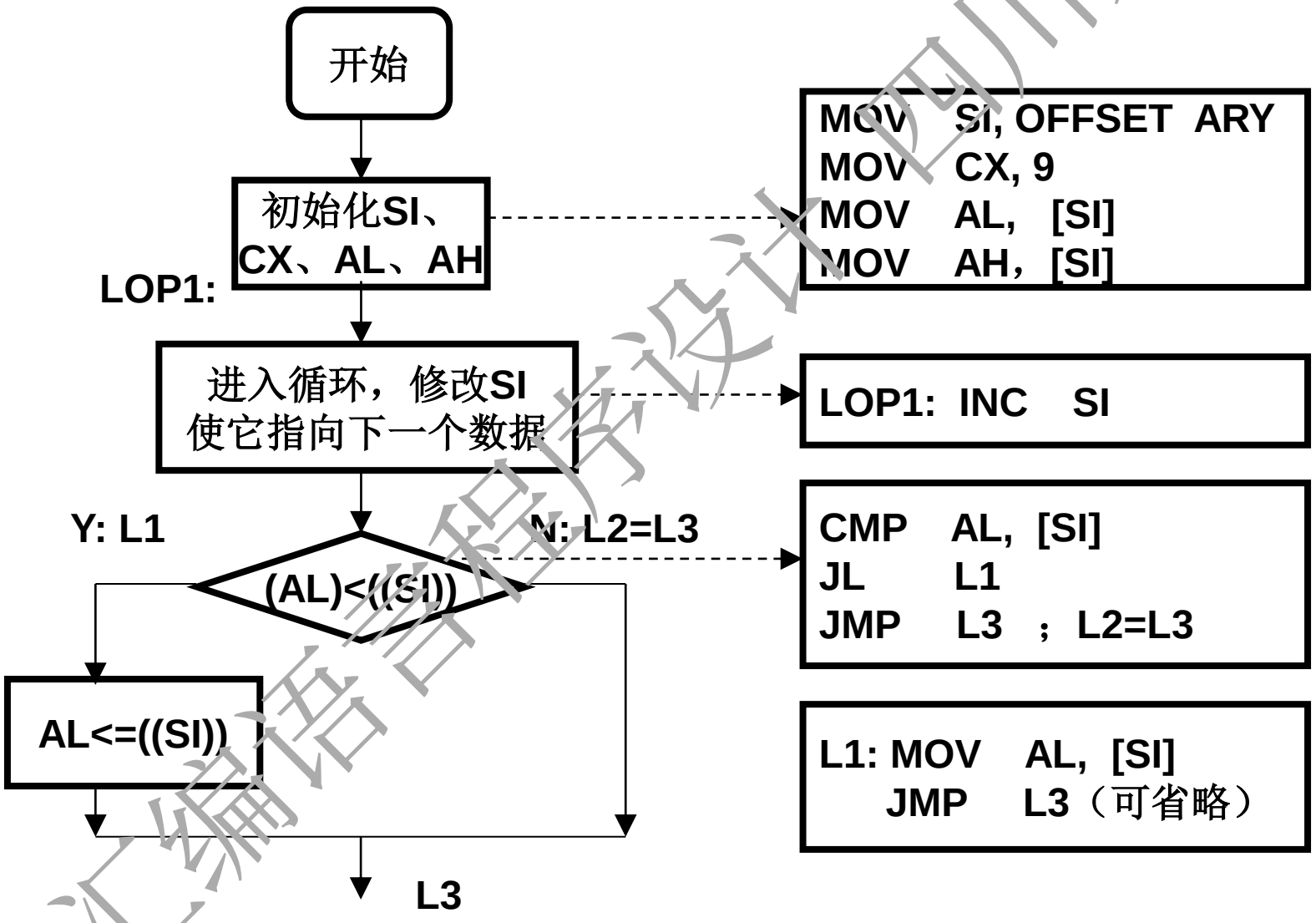
- 数据分析：
- ARY字节数组占用10个字节
- MIN字节单元用于存放最小值
- MAX字节单元用于存放最大值

（3）带符号数条件转移指令

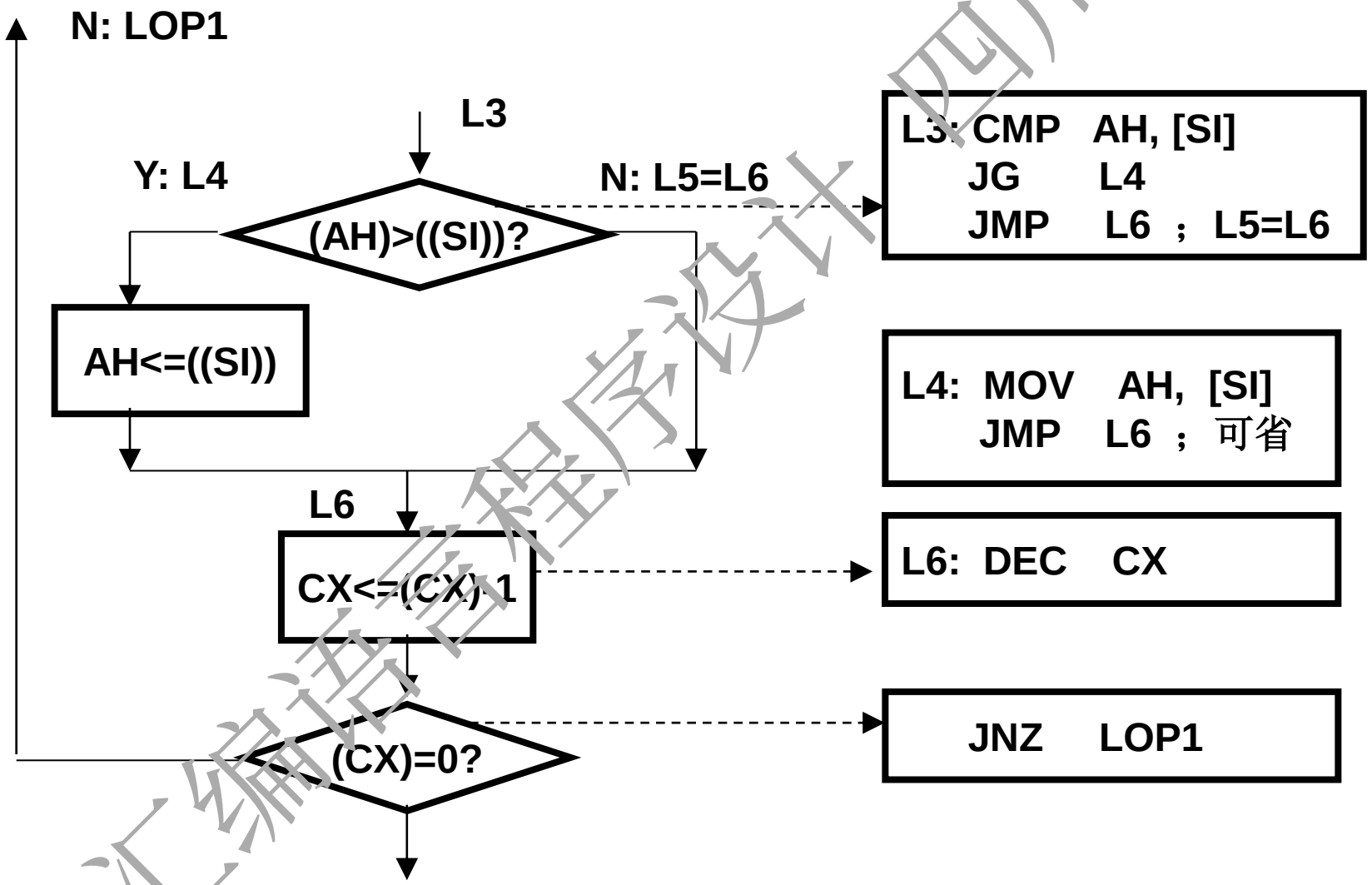
- 算法分析：
- 增加了求最小值的功能，可在求最大值的循环中一起完成，只是在循环中增加一个分支结构。

- 在前面的例子中使用AL寄存器存放当前比较的最大值，这里增加AH寄存器存放当前比较的最小值。

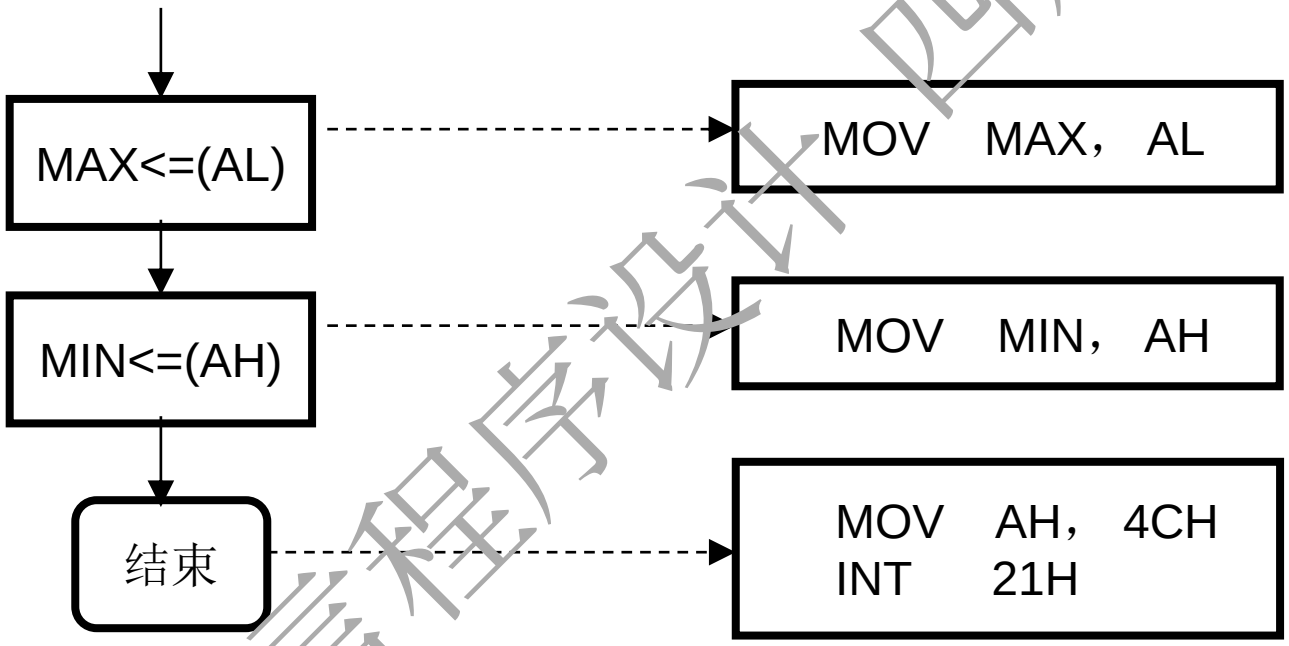
（3）带符号数条件转移指令



（3）带符号数条件转移指令



（3）带符号数条件转移指令



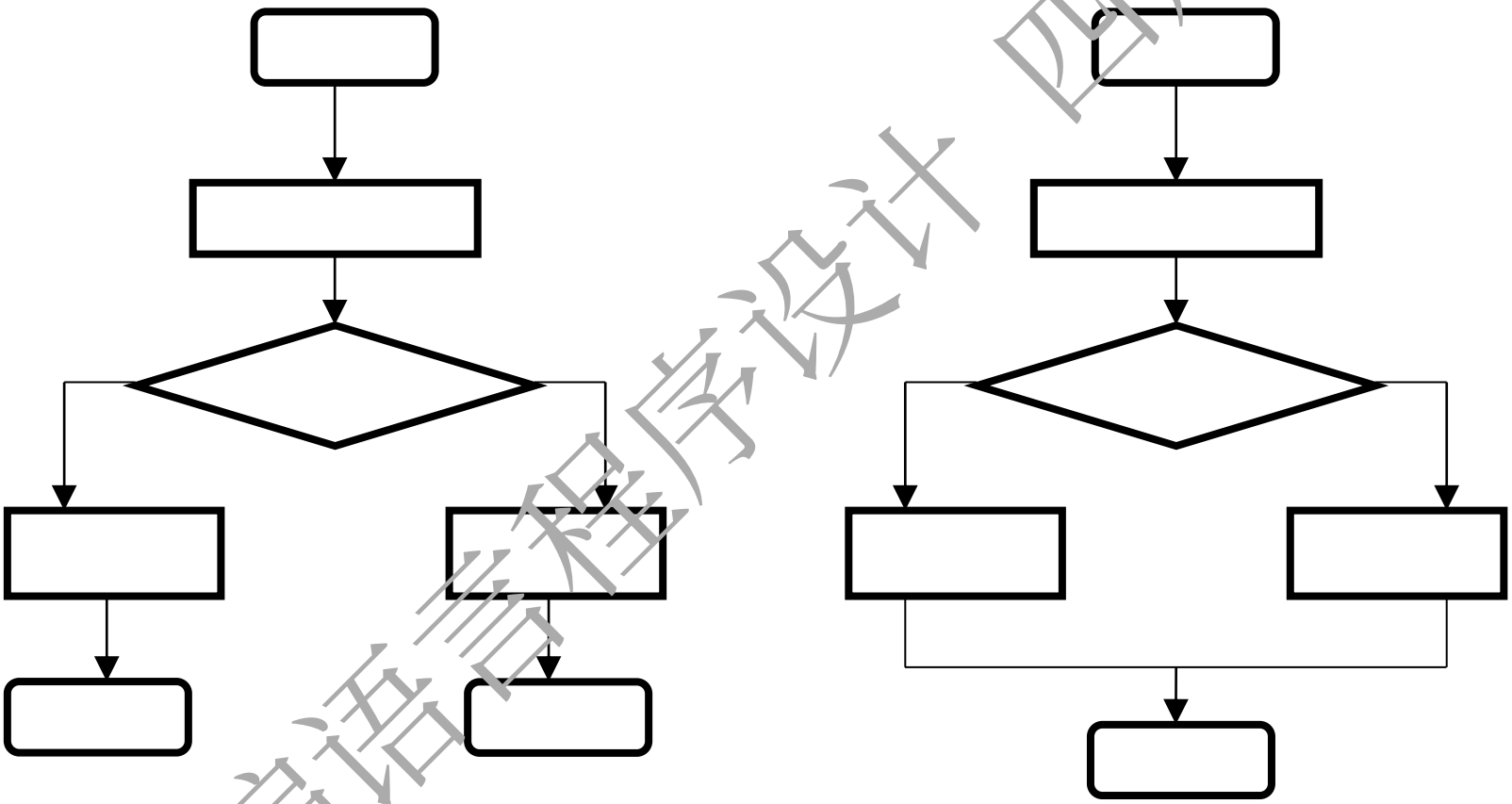
分支结构

- 分支结构是结合使用条件转移指令和无条件转移指令来构造的。分支结构分为两种：

- 1）IF-THEN-ELSE结构：双分支结构，把程序流程分为两支，这种结构和条件转移指令本来的特征是一致的。

- 2）CASE结构：多分支结构，可嵌套使用双分支结构来实现，但执行效率不高，也可通过建立多分支表的方法来建立效率较高的多分支结构。

结构化程序设计



结构化程序设计

- 在非结构化的分支结构中，每一个分支可能有多个出口，如果需要分析或调试这一分支结构，那么必须在源程序中找到所有的出口位置。

- 非结构化的分支会造成程序的可读性下降，调试难度加大。

- 如果采用结构化的分支，那么分支的所有出口都是一致的，程序的分析和调试都非常方便。

5.3 循环程序设计

- 所有循环结构均可使用条件转移指令来实现。

- 高级语言中的WHILE循环、DO UNTILL循环，都很适合使用条件转移指令实现。

5.3 循环程序设计

- FOR循环（记数循环）同样可以使用DEC指令和条件转移指令相结合来实现。

- 但是为了简化记数循环的设计，8086/8088系统提供了专门的记数循环指令。

5.3.1 循环控制指令

- 循环指令和转移指令的功能基本一致，都是完成程序流程的转移，区别在于循环指令会隐含使用CX寄存器作为记数器，每执行一次循环指令，（CX）就会自动被减1。

5.3.1 循环控制指令

- 机器指令格式：
- OPCODE（8位）　DISP（8位）

- DISP和条件转移指令中的位移量解释相同，作为相对位移量，转移范围也是-128到127。

5.3.1 循环控制指令

- 执行步骤：
- 1）CX<=（CX）-1
- 2）若循环条件满足，IP<=（IP）+DISP

- 循环条件主要是指（CX）是否为0，如果（CX）=0则停止循环，不改变程序的执行流程，否则执行流程转移，继续循环。

5.3.1 循环控制指令

- 有些循环指令在循环条件中还加上了对ZF标志的判断。

- 标志位影响：所有转移指令和循环指令都不会影响任何的标志位；

- 虽然它们可能会对标志位的当前取值进行判断。

（1）LOOP指令

- 格式：LOOP　目标地址

- 执行步骤：
- 1）CX<=（CX）-1
- 2）如果（CX）≠0，转移到目标地址，否则停止循环，顺序执行下一条指令

- 使用前提：在使用LOOP指令控制循环以前（进入程序的循环体以前），必须把循环次数保存到CX。

（1）LOOP指令

- 例：
- 将　dec cx
- 　　jnz lop

- 替换为
- 　　loop lop

（2）LOOPZ / LOOPE指令

- 格式：LOOPZ / LOOPE　目标地址

- 执行步骤：
- 1）CX<=（CX）-1
- 2）如果（CX）≠0 AND ZF=1，则转移到目标地址，否则停止循环，顺序执行下一条指令。

（2）LOOPZ / LOOPE指令

- 例：试编写一程序，在一字符串中查找第一个非空格字符，并将其在字符串中的序号（1~n）送入index单元中。如果未找到非空格字符，则将全1送到index单元中。

（2）LOOPZ / LOOPE指令

- data segment
- strg db ‘CHECK NO_SPACE’
- leng db $-strg
- index db ?
- data ends

- stack1 segment para stack
- dw 20h dup(0)
- stack1 ends

（2）LOOPZ / LOOPE指令

```
code  segment
        assume cs:code,ds:data,ss:stack1
start:  mov  ax, data
        mov  ds, ax
        mov  cx, leng
        mov  bx, -1
next:  inc    bx
        cmp  strg[bx], '
        loopz  next
        jnz   found
        mov  bl, 0feh
```

（2）LOOPZ / LOOPE指令

- found: inc   bl
- mov index, bl
- mov ah, 4ch
- int   21h
- code   ends
- end   start

（3）LOOPNE / LOOPNZ指令

- 格式：LOOPNE / LOOPNZ 目标地址

- 执行步骤：
- 1）CX<=（CX）-1
- 2）如果（CX）≠0 AND ZF=0，则转移到目标地址，否则停止循环，顺序执行下一条指令。

（3）LOOPNE / LOOPNZ指令

- 例：试编写程序，计算两个字节数组ary1和ary2对应元素之和，一直计算到两数之和为0或数组结束为止。并将和存入非0数组sum中，将该数组长度存放在num单元中。

（3）LOOPNE / LOOPNZ指令

- data segment
- ary1 db 12,10,3,5,-1,7,34,8,9,10
- ary2 db 14,23,6,-2,1,9,45,21,8,24
- leng equ ary2-ary1
- sum db leng dup(?)
- num db ?
- data ends

- stack1 segment para stack
- dw 20h dup(0)
- stack1 ends

（3）LOOPNE / LOOPNZ指令

```
coseg  segment
        assume cs:coseg,ds:data,ss:stack1
begin:  mov  ax, data
        mov  ds, ax
        mov  cx, leng
        mov  bx, -1
nzero:  inc   bx
        mov  al,  ary1[bx]
        add   al,  ary2[bx]
        mov  sum[bx], al
        loopnz  nzero
        jz    zero
```

（3）LOOPNE / LOOPNZ指令

- inc bl
- zero: mov num, bl
- mov ah, 4ch
- int 21h
- coseg ends
- end begin

（4）JCXZ在循环程序中的作用

- 进入循环体之前，首先使用JCXZ指令判断（CX）是否为0，如果为0就跳转到循环体出口，以免执行错误的循环过程。

```
        mov   cx, count
        jcxz    next
Lop:     ……
        loop   lop
next:    ……
```

5.3.2 单重循环程序设计

- （1）记数循环

- 对应高级语言中的FOR循环。

- （2）条件循环
- 对应高级语言中的WHILE或DO UNTIL循环。

5.3.2 单重循环程序设计

- 例 试编写一程序，统计字单元VARW中含“1”数据位的个数，统计结果存放于CONT单元中。

- 算法思想：将VARW中各数据位逐位移动到最高位进行判断，并计数。

- 条件循环：每次移位后判断是否为全0，如果是，则不需要继续移位和统计。

5.3.2 单重循环程序设计

- data segment
- varw dw 1101010010001000B
- cont db ?
- data ends

- stack1 segment stack
- dw 20h dup(0)
- stack1 ends

5.3.2 单重循环程序设计

```
code  segment
        assume  cs:code, ds:data, ss:stack1
begin: mov    ax, data
        mov    ds, ax
        mov    cl,  0
        mov    ax, varw
lop:    test    ax,  0ffffh
        jz      end0
```

5.3.2 单重循环程序设计

```
        jns     shift
        inc     cl
shift:  shl     ax,  1
        jmp     lop
end0:   mov     cont, cl
        mov     ah, 4ch
        int     21h
code    ends
        end     begin
```

5.3.3 多重循环程序设计

- 在程序中嵌套使用循环结构，称为多重循环结构。
- 和高级语言相比，汇编语言中多重循环结构设计相对烦琐。
- 需注意程序流程从外层循环进入内层循环时重新设置内层循环的初始环境。

5.3.3 多重循环程序设计

- 例 不使用乘法指令，试编写一程序，求级数$1^2+2^2+3^2+……$的前N项和。

- 算法思想：
- $N^2=N+N+…+N$（N次累加）；累加使用内层循环实现；级数和累加使用外层循环实现，总体上呈现双重循环结构。

5.3.3 多重循环程序设计

- data segment
- sum dw ?
- n db 20
- data ends

- stack1 segment stack
- dw 20h dup(0)
- stack1 ends

5.3.3 多重循环程序设计

```
code  segment
        assume  cs:code, ds:data, ss:stack1
start:  mov  ax, data
        mov  ds, ax
        mov  dx,  0
        mov  cx,  0
        mov  cl,  n
```

5.3.3 多重循环程序设计

```
lop1:  mov  ax, 0
       mov  bx, cx
lop2:  add   ax, cx
       dec   bx
       jnz   lop2
       add   dx, ax
       loop  lop1
```

5.3.3 多重循环程序设计

```
        mov  sum, dx
        mov  ah, 4ch
        int    21h
code  ends
        end  start
```